xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix Version 6 (v6). xv6 loosely follows the structure and style of v6, but is implemented for a modern x86-based multiprocessor using ANSI C.

ACKNOWLEDGMENTS

xv6 is inspired by John Lions's Commentary on UNIX 6th Edition (Peer to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14, 2000)). See also http://pdos.csail.mit.edu/6.828/2007/v6.html, which provides pointers to on-line resources for v6.

xv6 borrows code from the following sources:
JOS (asm.h, elf.h, mmu.h, bootasm.S, ide.c, console.c, and others)
Plan 9 (bootother.S, mp.h, mp.c, lapic.c)
FreeBSD (ioapic.c)
NetBSD (console.c)

The following people made contributions:
Russ Cox (context switching, locking)
Cliff Frey (MP)
Xiao Yu (MP)

The code in the files that constitute xv6 is
Copyright 2006-2007 Frans Kaashoek, Robert Morris, and Russ Cox.

ERROR REPORTS

If you spot errors or have suggestions for improvement, please send email to Frans Kaashoek and Robert Morris (kaashoek,rtm@csail.mit.edu).

BUILDING AND RUNNING XV6

To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run "make". On non-x86 or non-ELF machines (like OS X, even on x86), you will need to install a cross-compiler gcc suite capable of producing x86 ELF binaries. See http://pdos.csail.mit.edu/6.828/2007/tools.html. Then run "make TOOLPREFIX=i386-jos-elf-".

To run xv6, you can use Bochs or QEMU, both PC simulators. Bochs makes debugging easier, but QEMU is much faster. To run in Bochs, run "make bochs" and then type "c" at the bochs prompt. To run in QEMU, run "make qemu". Both log the xv6 screen output to standard output.

To create a typeset version of the code, run "make xv6.pdf". This requires the "mpage" text formatting utility. See http://www.mesa.nl/pub/mpage/.

The numbers to the left of the file names in the table are sheet numbers. The source code has been printed in a double column format with fifty lines per column, giving one hundred lines per sheet (or page). Thus there is a convenient relationship between line numbers and sheet numbers.

# basic headers
01 types.h
01 param.h
02 defs.h
04 x86.h
06 asm.h
06 mmu.h
08 elf.h

# startup
09 bootasm.S
10 bootother.S
11 bootmain.c
12 main.c

# locks
13 spinlock.h
13 spinlock.c

# processes
15 proc.h
16 proc.c
22 swtch.S
22 kalloc.c

system calls
24 traps.h
24 vectors.pl
25 trapasm.S
25 trap.c
27 syscall.h
27 syscall.c
29 sysproc.c

file system
30 buf.h
30 fcntl.h
31 stat.h
31 fs.h
32 file.h
33 ide.c
35 bio.c
36 fs.c
44 file.c
45 sysfile.c
50 exec.c

pipes
51 pipe.c

string operations
53 string.c

low-level hardware
54 mp.h
55 mp.c
56 lapic.c
58 ioapic.c
59 picirq.c
61 kbd.h
62 kbd.c
63 console.c
66 timer.c

user-level
67 initcode.S
67 usys.S
68 init.c
68 sh.c

The source listing is preceded by a cross-reference that lists every defined constant, struct, global variable, and function in xv6. Each entry gives, on the same line as the name, the line number (or, in a few cases, numbers) where the name is defined. Successive lines in an entry list the line numbers where the name is used. For example, this entry:

```
    swtch 2208
        0318 1928 1967 2207
        2208
```

indicates that swtch is defined on line 2208 and is mentioned on five lines on sheets 03, 19, and 22.

```
acquire 1373
    0321 1373 1377 1759
    1917 1975 2018 2033
    2066 2079 2123 2158
    2315 2362 2616 2971
    3407 3465 3570 3629
    3857 3890 3910 3939
    3954 3964 4425 4441
    4456 5213 5234 5255
    6360 6516 6558 6606
allocproc 1754
    1754 1807 1860
alltraps 2506
    2459 2467 2480 2485
    2505 2506
ALT 6110
    6110 6138 6140
argfd 4563
    4563 4606 4621 4633
    4644 4656
argint 2794
    0339 2794 2808 2824
    2931 2956 2969 4568
    4621 4633 4858 4921
    4922 4957
argptr 2804
    0340 2804 4621 4633
    4656 4982
argstr 2821
    0341 2821 4668 4758
    4858 4906 4920 4935
    4957
BACK 6861
    6861 6974 7120 7389
backcmd 6896 7114
    6896 6909 6975 7114
    7116 7242 7355 7390
BACKSPACE 6450
    6450 6467 6526 6532
balloc 3704
    3704 3725 4017 4025
    4029
BBLOCK 3191
    3191 3713 3739
bfree 3730
    3730 4062 4072 4075
bget 3566
    3566 3596 3606
binit 3539
    0210 1227 3539
bmap 4010
    4010 4036 4119 4169
    4222
bootmain 1116
    0976 1116
bootothers 1267
    1207 1234 1267
BPB 3188
    3188 3191 3712 3714
    3740
bread 3602
    0211 3602 3682 3693
    3713 3739 3811 3832
    3917 4026 4068 4119
    4169 4222
brelse 3624
    0212 3624 3627 3684
    3696 3719 3723 3746
    3817 3820 3841 3925
    4032 4074 4122 4173
    4233 4237
BSIZE 3158
    3158 3168 3182 3188
    3694 4119 4120 4121
    4165 4166 4169 4170
    4171 4221 4222 4224
buf 3000
    0200 0211 0212 0213
    0253 3000 3004 3005
    3006 3310 3325 3328
    3375 3404 3454 3456
    3459 3527 3531 3535
    3541 3553 3565 3568
    3601 3604 3614 3624
    3669 3680 3691 3707
    3732 3805 3829 3904
    4013 4057 4105 4155
    4215 6328 6339 6342
    6345 6503 6524 6537
    6568 6601 6608 6984
    6987 6988 6989 7003
    7015 7016 7019 7020
    7021 7025
bwrite 3614
    0213 3614 3617 3695
    3718 3745 3816 3840
    4030 4172
bzero 3689
    3689 3736
B_BUSY 3009
```

```
    3009 3458 3576 3577
    3588 3591 3616 3626
    3638
B_DIRTY 3011
    3011 3387 3416 3421
    3460 3479 3618
B_VALID 3010
    3010 3420 3460 3479
    3607
C 6131 6509
    6131 6179 6204 6205
    6206 6207 6208 6210
    6509 6519 6522 6529
    6539 6569
CAPSLOCK 6112
    6112 6145 6286
cgaputc 6455
    6455 6496
cli 0520
    0520 0522 0915 1029
    1460 6406 6490
cmd 6865
    6865 6877 6886 6887
    6892 6893 6898 6902
    6906 6915 6918 6923
    6931 6937 6941 6951
    6975 6977 7052 7055
    7057 7058 7059 7060
    7063 7064 7066 7068
    7069 7070 7071 7072
    7073 7074 7075 7076
    7079 7080 7082 7084
    7085 7086 7087 7088
    7089 7100 7101 7103
    7105 7106 7107 7108
    7109 7110 7113 7114
    7116 7118 7119 7120
    7121 7122 7212 7213
    7214 7215 7217 7221
    7224 7230 7231 7234
    7237 7239 7242 7246
    7248 7250 7253 7255
    7258 7260 7263 7264
    7275 7278 7281 7285
    7300 7303 7308 7312
    7313 7316 7321 7322
    7328 7337 7338 7344
    7345 7351 7352 7361
    7364 7366 7372 7373
    7378 7384 7390 7391
    7394
CONSOLE 3290
    3290 6621 6622
consoleinit 6616
    0216 1219 6616
consoleintr 6512
    0218 6298 6512
consoleread 6551
    6551 6622
consolewrite 6601
    6601 6621
consputc 6487
    6315 6345 6366 6384
    6387 6391 6392 6487
    6526 6532 6538 6608
context 1518
    0201 0318 1518 1537
    1559 1678 1787 1788
    1789 1790 1928 1967
cprintf 1221 6352
    0217 1221 1222 1258
    1262 1676 1680 1682
    2286 2375 2637 2653
    2658 2882 3410 5619
    5639 5761 5912 6352
    6408 6409 6410 6413
cpu 1557
    0256 1221 1222 1258
    1260 1262 1271 1306
    1365 1386 1408 1446
    1461 1462 1470 1472
    1557 1567 1571 1582
    1705 1710 1715 1724
    1725 1726 1727 1728
    1729 1928 1959 1966
    1967 1968 2615 2637
    2638 2653 2654 2658
    2659 5512 5513 5761
    6408
cpunum 5751
    0269 1255 1256 1279
    1707 5751 5923 5932
CR0_PE 0910 1024
    0956 1056
create 4801
    4801 4821 4834 4838
    4862 4906 4923
CRTPORT 6451
    6451 6460 6461 6462
    6463 6479 6480 6481
    6482
CTL 6109
    6109 6135 6139 6285
devsw 3283
    3283 3288 4108 4110
    4158 4160 4407 6621
    6622
dinode 3172
    3172 3182 3806 3812
    3830 3833 3905 3918
dirent 3203
    3203 4216 4223 4224
    4255 4705 4754
dirlink 4252
    0234 4252 4267 4275
    4684 4833 4837 4838
dirlookup 4212
    0235 4212 4219 4259
    4374 4770 4811
DIRSIZ 3201
    3201 3205 4205 4272
    4328 4329 4391 4665
    4755 4805
DPL_USER 0711
    0711 1724 1725 1817
    1818 2572 2666 2675
E0ESC 6116
    6116 6270 6274 6275
    6277 6280
elfhdr 0855
    0855 1118 1123 5014
ELF_MAGIC 0852
    0852 1129 5028
ELF_PROG_LOAD 0886
    0886 5036 5067
EOI 5663
    5663 5734 5775
ERROR 5681
    5681 5727
ESR 5666
    5666 5730 5731
EXEC 6857
    6857 6922 7059 7365
exec 5009
    0222 4972 5009 6768
    6829 6830 6926 6927
execcmd 6869 7053
    6869 6910 6923 7053
    7055 7321 7327 7328
    7356 7366
exit 2104
    0302 2104 2140 2605
    2609 2667 2676 2916
    6715 6718 6761 6826
    6831 6916 6925 6935
    6980 7028 7035
fdalloc 4582
    4582 4608 4874 4987
fetchint 2766
    0342 2766 2796 4963
fetchstr 2778
    0343 2778 2826 4969
file 3250
    0202 0225 0226 0227
    0229 0230 0231 0287
    1540 3250 3671 4404
    4410 4420 4423 4426
    4438 4439 4452 4454
    4476 4502 4522 4557
    4563 4566 4582 4603
    4617 4629 4642 4653
    4855 4979 5156 5171
    6310 6878 6933 6934
    7064 7072 7272
filealloc 4421
    0225 4421 4874 5177
fileclose 4452
    0226 2115 4452 4458
    4647 4876 4990 4991
    5204 5206
filedup 4439
    0227 1880 4439 4443
    4610
fileinit 4414
    0228 1228 4414
fileread 4502
    0229 4502 4517 4623
filestat 4476
    0230 4476 4658
filewrite 4522
    0231 4522 4537 4635
FL_IF 0660
    0660 1462 1468 1821
    1963 5758
fork 1854
    0303 1854 2910 6760
    6823 6825 7043 7045
fork1 7039
    6900 6942 6954 6961
    6976 7024 7039
```

```
forkret 1984
    1616 1790 1984
gatedesc 0801
    0464 0467 0801 2560
getcallerpcs 1426
    0322 1387 1426 1678
    6411
getcmd 6984
    6984 7015
gettoken 7156
    7156 7241 7245 7257
    7270 7271 7307 7311
    7333
growproc 1834
    0304 1834 2959
havedisk1 3327
    3327 3364 3462
holding 1444
    0323 1376 1404 1444
    1957
ialloc 3802
    0236 3802 3822 4820
    4821
IBLOCK 3185
    3185 3811 3832 3917
ICRHI 5674
    5674 5737 5807 5819
ICRLO 5667
    5667 5738 5739 5808
    5810 5820
ID 5660
    5660 5693 5766
ideinit 3351
    0251 1230 3351
ideintr 3402
    0252 2624 3402
idelock 3324
    3324 3355 3407 3409
    3428 3465 3480 3482
iderw 3454
    0253 3454 3459 3461
    3608 3619
idestart 3375
    3328 3375 3378 3426
    3475
idewait 3332
    3332 3358 3380 3416
IDE_BSY 3312
    3312 3336
IDE_CMD_READ 3317
    3317 3391
IDE_CMD_WRITE 3318
    3318 3388
IDE_DF 3314
    3314 3338
IDE_DRDY 3313
    3313 3336
IDE_ERR 3315
    3315 3338
idtinit 2578
    0351 1259 2578
idup 3888
    0237 1881 3888 4361
iget 3853
    3794 3818 3853 3873
    4234 4359
iinit 3789
    0238 1229 3789
ilock 3902
    0239 3902 3908 3928
    4364 4479 4511 4531
    4672 4683 4693 4762
    4774 4809 4813 4823
    4867 4937 5023 6563
    6583 6610
inb 0403
    0403 0928 0936 1154
    3336 3363 5647 6264
    6267 6461 6463
initlock 1361
    0324 1361 1622 2283
    2574 3355 3543 3791
    4416 5185 6618 6619
inode 3263
    0203 0234 0235 0236
    0237 0239 0240 0241
    0242 0243 0245 0246
    0247 0248 0249 1541
    3256 3263 3284 3285
    3674 3785 3794 3801
    3827 3852 3855 3861
    3887 3888 3902 3934
    3952 3974 4010 4054
    4085 4102 4152 4211
    4212 4252 4256 4353
    4356 4388 4395 4666
    4702 4753 4800 4804
    4856 4904 4915 4933
    5015 6551 6601
INPUT_BUF 6500
```

```
    6500 6503 6524 6536
    6537 6539 6568
insl 0412
    0412 0414 1173 3417
INT_DISABLED 5869
    5869 5917
IOAPIC 5858
    5858 5908
ioapic 5877
    5607 5629 5630 5874
    5877 5886 5887 5893
    5894 5908
ioapicenable 5923
    0256 3357 5923 6626
ioapicid 5516
    0257 5516 5630 5911
    5912
ioapicinit 5901
    0258 1218 5901 5912
ioapicread 5884
    5884 5909 5910
ioapicwrite 5891
    5891 5917 5918 5931
    5932
IO_PIC1 5957
    5957 5970 5985 5994
    5997 6002 6012 6026
    6027
IO_PIC2 5958
    5958 5971 5986 6015
    6016 6017 6020 6029
    6030
IO_RTC 5786
    5786 5799 5800
IO_TIMER1 6659
    6659 6668 6678 6679
IPB 3182
    3182 3185 3191 3812
    3833 3918
iput 3952
    0240 2120 3952 3958
    3977 4260 4382 4471
    4689 4943
IRQ_COM1 2433
    2433 2631
IRQ_ERROR 2435
    2435 5727
IRQ_IDE 2434
    2434 2623 3356 3357
IRQ_KBD 2432
    2432 2627 6625 6626
IRQ_SLAVE 5960
    5960 5964 6002 6017
IRQ_SPURIOUS 2436
    2436 2636 5707
IRQ_TIMER 2431
    2431 2614 2671 5714
    6680
isdirempty 4702
    4702 4709 4778
ismp 5514
    0277 1231 5514 5612
    5905 5925
itrunc 4054
    3674 3961 4054
iunlock 3934
    0241 3934 3937 3976
    4371 4481 4514 4534
    4679 4880 4942 6556
    6605
iunlockput 3974
    0242 3974 4366 4375
    4378 4674 4685 4688
    4696 4766 4771 4779
    4780 4791 4795 4812
    4816 4840 4869 4877
    4908 4925 4939 5077
    5118
iupdate 3827
    0243 3827 3963 4080
    4178 4678 4695 4789
    4794 4827 4831
I_BUSY 3277
    3277 3911 3913 3936
    3940 3957 3959
I_VALID 3278
    3278 3916 3926 3955
kalloc 2354
    0261 1283 1772 1812
    1838 1865 2354 2360
    2375 5058 5179
KBDATAP 6104
    6104 6267
kbdgetc 6256
    6256 6298
kbdintr 6296
    0266 2628 6296
KBSTATP 6102
    6102 6264
KBS_DIB 6103
```

```
    6103 6265
KEY_DEL 6128
    6128 6169 6191 6215
KEY_DN 6122
    6122 6165 6187 6211
KEY_END 6120
    6120 6168 6190 6214
KEY_HOME 6119
    6119 6168 6190 6214
KEY_INS 6127
    6127 6169 6191 6215
KEY_LF 6123
    6123 6167 6189 6213
KEY_PGDN 6126
    6126 6166 6188 6212
KEY_PGUP 6125
    6125 6166 6188 6212
KEY_RT 6124
    6124 6167 6189 6213
KEY_UP 6121
    6121 6165 6187 6211
kfree 2305
    0262 1843 1866 2169
    2170 2287 2305 2310
    5107 5117 5202 5223
kill 2075
    0305 2075 2658 2933
    6767
kinit 2277
    0263 1224 2277
ksegment 1703
    0309 1216 1257 1703
KSTACKSIZE 0152
    0152 1283 1284 1729
    1772 1776 1866 2170
lapiceoi 5772
    0271 2621 2625 2629
    2633 2639 5772
lapicinit 5701
    0272 1215 1256 5701
lapicstartap 5791
    0273 1286 5791
lapicw 5690
    5690 5707 5713 5714
    5715 5718 5719 5724
    5727 5730 5731 5734
    5737 5738 5743 5775
    5807 5808 5810 5819
    5820
lgdt 0453
    0453 0461 0954 1054
    1711
lidt 0467
    0467 0475 2580
LINT0 5679
    5679 5718
LINT1 5680
    5680 5719
LIST 6860
    6860 6940 7107 7383
listcmd 6890 7101
    6890 6911 6941 7101
    7103 7246 7357 7384
loadgs 0514
    0514 1712
ltr 0479
    0479 0481 1730
MAXARGS 6863
    6863 6871 6872 7340
MAXFILE 3169
    3169 4165 4166
memcmp 5311
    0330 5311 5543 5588
memmove 5327
    0331 1276 1814 1841
    1871 3683 3839 3924
    4121 4171 4329 4331
    5088 5327 6474
memset 5304
    0332 1789 1813 1816
    1842 2313 3694 3814
    4784 4959 5061 5075
    5304 6476 6987 7058
    7069 7085 7106 7119
microdelay 5781
    0274 5781 5809 5811
    5821
min 3673
    3673 4120 4170
mp 5402
    5402 5507 5536 5542
    5543 5544 5555 5560
    5564 5565 5568 5569
    5580 5583 5585 5587
    5594 5604 5610 5643
mpbcpu 5519
    0278 1215 1255 5519
MPBUS 5452
    5452 5633
mpconf 5413
```

```
    5413 5579 5582 5587
    5605
mpconfig 5580
    5580 5610
mpinit 5601
    0279 1214 5601 5619
    5620 5639 5640
MPIOAPIC 5453
    5453 5628
mpioapic 5439
    5439 5607 5629 5631
MPIOINTR 5454
    5454 5634
MPLINTR 5455
    5455 5635
mpmain 1253
    1208 1237 1253 1258
    1285
MPPROC 5451
    5451 5616
mpproc 5428
    5428 5606 5617 5626
mpsearch 5556
    5556 5585
mpsearch1 5537
    5537 5564 5568 5571
namecmp 4203
    0244 4203 4228 4765
namei 4389
    0245 1826 4389 4670
    4865 4935 5021
nameiparent 4396
    0246 4354 4369 4381
    4396 4681 4760 4807
namex 4354
    4354 4392 4398
NBUF 0156
    0156 3531 3553
NCPU 0153
    0153 1571 5512
ncpu 5515
    1222 1278 1572 3357
    5515 5618 5619 5623
    5624 5625
NDEV 0158
    0158 4108 4158 4407
NDIRECT 3167
    3167 3169 3178 3274
    4015 4020 4024 4025
    4060 4067 4068 4075
```

```
    4076
NELEM 0362
    0362 1672 2879 4961
nextpid 1615
    1615 1768
NFILE 0155
    0155 4410 4426
NINDIRECT 3168
    3168 3169 4022 4070
NINODE 0157
    0157 3785 3861
NO 6106
    6106 6152 6155 6157
    6158 6159 6160 6162
    6174 6177 6179 6180
    6181 6182 6184 6202
    6203 6205 6206 6207
    6208
NOFILE 0154
    0154 1540 1878 2113
    4570 4586
NPROC 0150
    0150 1610 1669 1760
    1918 2057 2080 2129
    2162
NSEGS 1508
    1508 1561
nulterminate 7352
    7215 7230 7352 7373
    7379 7380 7385 7386
    7391
NUMLOCK 6113
    6113 6146
outb 0421
    0421 0933 0941 1164
    1165 1166 1167 1168
    1169 3361 3370 3381
    3382 3383 3384 3385
    3386 3388 3391 5646
    5647 5799 5800 5970
    5971 5985 5986 5994
    5997 6002 6012 6015
    6016 6017 6020 6026
    6027 6029 6030 6460
    6462 6479 6480 6481
    6482 6677 6678 6679
outsl 0433
    0433 0435 3389
outw 0427
    0427 0982 0984 1082
```

```
    1084
O_CREATE 3053
    3053 4861 7278 7281
O_RDONLY 3050
    3050 4868 7275
O_RDWR 3052
    3052 4886 6814 6816
    7007
O_WRONLY 3051
    3051 4885 4886 7278
    7281
PAGE 0151
    0151 0152 1811 2284
    2285 2309 2359 5054
    5057 5179 5202 5223
panic 6401 7032
    0219 1377 1405 1469
    1471 1958 1960 1962
    1964 2006 2009 2110
    2140 2310 2321 2360
    2655 3378 3459 3461
    3463 3596 3617 3627
    3725 3743 3822 3873
    3908 3928 3937 3958
    4036 4219 4267 4275
    4443 4458 4517 4537
    4709 4777 4786 4821
    4834 4838 5620 5640
    6401 6408 6901 6920
    6953 7032 7045 7228
    7272 7306 7310 7336
    7341
panicked 6317
    6317 6414 6489
parseblock 7301
    7301 7306 7325
parsecmd 7218
    6902 7025 7218
parseexec 7317
    7214 7255 7317
parseline 7235
    7212 7224 7235 7246
    7308
parsepipe 7251
    7213 7239 7251 7258
parseredirs 7264
    7264 7312 7331 7342
PCINT 5678
    5678 5724
peek 7201
```

```
    7201 7225 7240 7244
    7256 7269 7305 7309
    7324 7332
picenable 5975
    0283 3356 5975 6625
    6680
picinit 5982
    0284 1217 5982
picsetmask 5967
    5967 5977 6033
pinit 1620
    0306 1225 1620
PIPE 6859
    6859 6950 7086 7377
pipe 5161
    0204 0288 0289 0290
    3255 4469 4509 4529
    5161 5173 5179 5185
    5189 5193 5211 5230
    5251 6763 6952 6953
pipealloc 5171
    0287 4984 5171
pipeclose 5211
    0288 4469 5211
pipecmd 6884 7080
    6884 6912 6951 7080
    7082 7258 7358 7378
piperead 5251
    0289 4509 5251
PIPESIZE 5159
    5159 5163 5236 5244
    5266
pipewrite 5230
    0290 4529 5230
popcli 1466
    0327 1421 1466 1469
    1471 1731
printint 6325
    6325 6374 6378
proc 1529
    0205 0301 0342 0343
    1204 1357 1529 1535
    1568 1583 1605 1610
    1613 1665 1669 1716
    1724 1725 1729 1753
    1756 1760 1804 1838
    1841 1842 1843 1844
    1845 1857 1864 1871
    1872 1873 1879 1880
    1881 1910 1918 1925
    1928 1932 1961 1967
    1976 2005 2023 2024
    2028 2055 2057 2077
    2080 2106 2109 2114
    2115 2116 2120 2121
    2126 2129 2130 2138
    2155 2162 2163 2182
    2188 2554 2604 2606
    2608 2651 2658 2659
    2660 2666 2671 2675
    2754 2766 2778 2796
    2810 2812 2826 2878
    2880 2883 2884 2905
    2939 2958 2974 3306
    3667 4361 4555 4570
    4587 4588 4646 4943
    4944 4963 4969 4989
    5003 5104 5107 5108
    5109 5110 5111 5154
    5237 5257 5510 5606
    5617 5618 5619 5622
    6312 6561
procdump 1654
    0307 1654 6520
proghdr 0874
    0874 1119 1133 5016
pushcli 1455
    0326 1375 1455 1723
readeflags 0485
    0485 1459 1468 1963
    5758
readi 4102
    0247 4102 4266 4512
    4708 4709 5026 5034
    5065 5073
readsb 3678
    3678 3711 3738 3809
readsect 1160
    1160 1195
readseg 1179
    1113 1126 1137 1179
REDIR 6858
    6858 6930 7070 7371
redircmd 6875 7064
    6875 6913 6931 7064
    7066 7275 7278 7281
    7359 7372
REG_ID 5860
    5860 5910
REG_TABLE 5862
    5862 5917 5918 5931
    5932
REG_VER 5861
    5861 5909
release 1402
    0325 1402 1405 1763
    1769 1934 1978 1987
    2019 2032 2068 2086
    2090 2176 2183 2343
    2369 2373 2619 2975
    2980 3409 3428 3482
    3578 3592 3641 3864
    3880 3892 3914 3942
    3960 3969 4429 4433
    4445 4460 4466 5222
    5225 5238 5247 5258
    5269 6398 6547 6562
    6582 6609
ROOTDEV 0159
    0159 4359
ROOTINO 3157
    3157 4359
run 2262
    1661 2262 2263 2269
    2307 2316 2317 2319
    2357
runcmd 6906
    6906 6920 6937 6943
    6945 6959 6966 6977
    7025
RUNNING 1526
    1526 1661 1927 1961
    2671
safestrcpy 5375
    0333 1825 5104 5375
sched 1953
    1953 1958 1960 1962
    1964 1977 2025 2139
scheduler 1908
    0308 1263 1559 1908
    1928 1967
SCROLLLOCK 6114
    6114 6147
SECTSIZE 1111
    1111 1173 1186 1189
    1194
SEG 0701
    0701 1708 1709 1710
    1724 1725
SEG16 0706
```

```
    0706 1726
segdesc 0677
    0450 0453 0677 0701
    0706 1561
SEG_ASM 0608
    0608 0992 0993 1092
    1093
SEG_KCODE 0907 1021 1502 2500
    0961 1061 1502 1708
    2571 2572
SEG_KCPU 1504 2502
    1504 1710 1712 2518
SEG_KDATA 0908 1022 1503 2501
    0966 1066 1503 1709
    1728 2515
SEG_NULLASM 0604
    0604 0991 1091
SEG_TSS 1507
    1507 1726 1727 1730
SEG_UCODE 1505
    1505 1724 1817
SEG_UDATA 1506
    1506 1725 1818
SETGATE 0821
    0821 2571 2572
SHIFT 6108
    6108 6136 6137 6285
skipelem 4315
    4315 4363
sleep 2003
    0311 1659 2003 2006
    2009 2188 2978 3480
    3581 3912 5242 5261
    6566 6779
spinlock 1301
    0206 0311 0321 0323
    0324 0325 0354 1301
    1358 1361 1373 1402
    1444 1606 1609 2003
    2260 2268 2557 2562
    3309 3324 3526 3530
    3668 3784 4405 4409
    5157 5162 6308 6320
    6502
start 0914 1028 6707
    0913 0914 0975 1027
    1028 1075 1076 6706
    6707
stat 3104
    0207 0230 0248 3104
    3665 4085 4476 4553
    4654 6803
stati 4085
    0248 4085 4480
STA_R 0617 0718
    0617 0718 0992 1092
    1708 1724
STA_W 0616 0717
    0616 0717 0993 1093
    1709 1710 1725
STA_X 0613 0714
    0613 0714 0992 1092
    1708 1724
sti 0526
    0526 0528 1473 1914
stosb 0442
    0442 0444 1139 5306
strlen 5389
    0334 5046 5086 5389
    7019 7223
strncmp 5351
    0335 4205 5351
strncpy 5361
    0336 4272 5361
STS_IG32 0732
    0732 0827
STS_T32A 0729
    0729 1726
STS_TG32 0733
    0733 0827
sum 5525
    5525 5527 5529 5531
    5532 5543 5592
superblock 3161
    3161 3678 3708 3733
    3807
SVR 5664
    5664 5707
swtch 2208
    0318 1928 1967 2207
    2208
SYSCALL 6753 6760 6761 6762 6763 67
    6760 6761 6762 6763
    6764 6765 6766 6767
    6768 6769 6770 6771
    6772 6773 6774 6775
    6776 6777 6778 6779
syscall 2874
    0344 2607 2756 2874
SYS_chdir 2716
```

```
    2716 2851
sys_chdir 4930
    2829 2851 4930
SYS_close 2707
    2707 2852
sys_close 4639
    2830 2852 4639
SYS_dup 2717
    2717 2853
sys_dup 4601
    2831 2853 4601
SYS_exec 2709
    2709 2854 6711
sys_exec 4951
    2832 2854 4951
SYS_exit 2702
    2702 2855 6716
sys_exit 2914
    2833 2855 2914
SYS_fork 2701
    2701 2856
sys_fork 2908
    2834 2856 2908
SYS_fstat 2713
    2713 2857
sys_fstat 4651
    2835 2857 4651
SYS_getpid 2718
    2718 2858
sys_getpid 2937
    2836 2858 2937
SYS_kill 2708
    2708 2859
sys_kill 2927
    2837 2859 2927
SYS_link 2714
    2714 2860
sys_link 4663
    2838 2860 4663
SYS_mkdir 2715
    2715 2861
sys_mkdir 4901
    2839 2861 4901
SYS_mknod 2711
    2711 2862
sys_mknod 4913
    2840 2862 4913
SYS_open 2710
    2710 2863
sys_open 4851
    2841 2863 4851
SYS_pipe 2704
    2704 2864
sys_pipe 4976
    2842 2864 4976
SYS_read 2706
    2706 2865
sys_read 4615
    2843 2865 4615
SYS_sbrk 2719
    2719 2866
sys_sbrk 2951
    2844 2866 2951
SYS_sleep 2720
    2720 2867
sys_sleep 2965
    2845 2867 2965
SYS_unlink 2712
    2712 2868
sys_unlink 4751
    2846 2868 4751
SYS_wait 2703
    2703 2869
sys_wait 2921
    2847 2869 2921
SYS_write 2705
    2705 2870
sys_write 4627
    2848 2870 4627
taskstate 0751
    0751 1560
TDCR 5685
    5685 5713
ticks 2563
    0352 2563 2617 2618
    2972 2973 2978
tickslock 2562
    0354 2562 2574 2616
    2619 2971 2975 2978
    2980
TICR 5683
    5683 5715
TIMER 5675
    5675 5714
timerinit 6674
    0347 1232 6674
TIMER_16BIT 6671
    6671 6677
TIMER_DIV 6666
    6666 6678 6679
```

```
TIMER_FREQ 6665
    6665 6666
TIMER_MODE 6668
    6668 6677
TIMER_RATEGEN 6670
    6670 6677
TIMER_SEL0 6669
    6669 6677
TPR 5662
    5662 5743
trap 2601
    2452 2454 2524 2601
    2653 2655 2658
trapframe 0552
    0552 1536 1780 2601
trapret 2529
    1617 1785 2528 2529
tvinit 2566
    0353 1226 2566
T_DEV 3102
    3102 4107 4157 4923
T_DIR 3100
    3100 4218 4365 4673
    4778 4787 4829 4868
    4906 4938
T_FILE 3101
    3101 4814 4862
T_IRQ0 2429
    2429 2614 2623 2627
    2631 2635 2636 2671
    5707 5714 5727 5917
    5931 5997 6016
T_SYSCALL 2426
    2426 2572 2603 6712
    6717 6757
usegment 1721
    0310 1721 1846 1926
    5112
userinit 1802
    0312 1233 1802
VER 5661
    5661 5723
wait 2153
    0313 2153 2923 6762
    6833 6944 6970 6971
    7026
waitdisk 1151
    1151 1163 1172
wakeup 2064
    0314 2064 2618 3422
    3639 3941 3966 5216
    5219 5241 5246 5268
    6541
wakeup1 2053
    2053 2067 2126 2133
writei 4152
    0249 4152 4274 4532
    4785 4786
xchg 0501
    0501 1260 1382 1419
yield 1973
    0315 1973 2672
```

```
typedef unsigned int   uint;
typedef unsigned short ushort;
typedef unsigned char  uchar;
```

```
#define NPROC        64  // maximum number of processes
#define PAGE       4096  // granularity of user-space memory allocation
#define KSTACKSIZE PAGE  // size of per-process kernel stack
#define NCPU          8  // maximum number of CPUs
#define NOFILE       16  // open files per process
#define NFILE       100  // open files per system
#define NBUF         10  // size of disk block cache
#define NINODE       50  // maximum number of active i-nodes
#define NDEV         10  // maximum major device number
#define ROOTDEV       1  // device number of file system root disk
```

```
struct buf;
struct context;
struct file;
struct inode;
struct pipe;
struct proc;
struct spinlock;
struct stat;

// bio.c
void            binit(void);
struct buf*     bread(uint, uint);
void            brelse(struct buf*);
void            bwrite(struct buf*);

// console.c
void            consoleinit(void);
void            cprintf(char*, ...);
void            consoleintr(int(*)(void));
void            panic(char*) __attribute__((noreturn));

// exec.c
int             exec(char*, char**);

// file.c
struct file*    filealloc(void);
void            fileclose(struct file*);
struct file*    filedup(struct file*);
void            fileinit(void);
int             fileread(struct file*, char*, int n);
int             filestat(struct file*, struct stat*);
int             filewrite(struct file*, char*, int n);

// fs.c
int             dirlink(struct inode*, char*, uint);
struct inode*   dirlookup(struct inode*, char*, uint*);
struct inode*   ialloc(uint, short);
struct inode*   idup(struct inode*);
void            iinit(void);
void            ilock(struct inode*);
void            iput(struct inode*);
void            iunlock(struct inode*);
void            iunlockput(struct inode*);
void            iupdate(struct inode*);
int             namecmp(const char*, const char*);
struct inode*   namei(char*);
struct inode*   nameiparent(char*, char*);
int             readi(struct inode*, char*, uint, uint);
void            stati(struct inode*, struct stat*);
int             writei(struct inode*, char*, uint, uint);
```

```
// ide.c
void            ideinit(void);
void            ideintr(void);
void            iderw(struct buf*);

// ioapic.c
void            ioapicenable(int irq, int cpu);
extern uchar    ioapicid;
void            ioapicinit(void);

// kalloc.c
char*           kalloc(int);
void            kfree(char*, int);
void            kinit(void);

// kbd.c
void            kbdintr(void);

// lapic.c
int             cpunum(void);
extern volatile uint*    lapic;
void            lapiceoi(void);
void            lapicinit(int);
void            lapicstartap(uchar, uint);
void            microdelay(int);

// mp.c
extern int      ismp;
int             mpbcpu(void);
void            mpinit(void);
void            mpstartthem(void);

// picirq.c
void            picenable(int);
void            picinit(void);

// pipe.c
int             pipealloc(struct file**, struct file**);
void            pipeclose(struct pipe*, int);
int             piperead(struct pipe*, char*, int);
int             pipewrite(struct pipe*, char*, int);
```

```
// proc.c
struct proc*    copyproc(struct proc*);
void            exit(void);
int             fork(void);
int             growproc(int);
int             kill(int);
void            pinit(void);
void            procdump(void);
void            scheduler(void) __attribute__((noreturn));
void            ksegment(void);
void            usegment(void);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(void);
void            wakeup(void*);
void            yield(void);

// swtch.S
void            swtch(struct context**, struct context*);

// spinlock.c
void            acquire(struct spinlock*);
void            getcallerpcs(void*, uint*);
int             holding(struct spinlock*);
void            initlock(struct spinlock*, char*);
void            release(struct spinlock*);
void            pushcli();
void            popcli();

// string.c
int             memcmp(const void*, const void*, uint);
void*           memmove(void*, const void*, uint);
void*           memset(void*, int, uint);
char*           safestrcpy(char*, const char*, int);
int             strlen(const char*);
int             strncmp(const char*, const char*, uint);
char*           strncpy(char*, const char*, int);

// syscall.c
int             argint(int, int*);
int             argptr(int, char**, int);
int             argstr(int, char**);
int             fetchint(struct proc*, uint, int*);
int             fetchstr(struct proc*, uint, char**);
void            syscall(void);

// timer.c
void            timerinit(void);
```

```
// trap.c
void            idtinit(void);
extern int      ticks;
void            tvinit(void);
extern struct spinlock tickslock;

// uart.c
void            uartinit(void);
void            uartintr(void);
void            uartputc(int);

// number of elements in fixed-size array
#define NELEM(x) (sizeof(x)/sizeof((x)[0]))
```

```c
// Routines to let C code use special x86 instructions.

static inline uchar
inb(ushort port)
{
  uchar data;

  asm volatile("in %1,%0" : "=a" (data) : "d" (port));
  return data;
}

static inline void
insl(int port, void *addr, int cnt)
{
  asm volatile("cld; rep insl" :
               "=D" (addr), "=c" (cnt) :
               "d" (port), "0" (addr), "1" (cnt) :
               "memory", "cc");
}

static inline void
outb(ushort port, uchar data)
{
  asm volatile("out %0,%1" : : "a" (data), "d" (port));
}

static inline void
outw(ushort port, ushort data)
{
  asm volatile("out %0,%1" : : "a" (data), "d" (port));
}

static inline void
outsl(int port, const void *addr, int cnt)
{
  asm volatile("cld; rep outsl" :
               "=S" (addr), "=c" (cnt) :
               "d" (port), "0" (addr), "1" (cnt) :
               "cc");
}

static inline void
stosb(void *addr, int data, int cnt)
{
  asm volatile("cld; rep stosb" :
               "=D" (addr), "=c" (cnt) :
               "0" (addr), "1" (cnt), "a" (data) :
               "memory", "cc");
}
```

```c
struct segdesc;

static inline void
lgdt(struct segdesc *p, int size)
{
  volatile ushort pd[3];

  pd[0] = size-1;
  pd[1] = (uint)p;
  pd[2] = (uint)p >> 16;

  asm volatile("lgdt (%0)" : : "r" (pd));
}

struct gatedesc;

static inline void
lidt(struct gatedesc *p, int size)
{
  volatile ushort pd[3];

  pd[0] = size-1;
  pd[1] = (uint)p;
  pd[2] = (uint)p >> 16;

  asm volatile("lidt (%0)" : : "r" (pd));
}

static inline void
ltr(ushort sel)
{
  asm volatile("ltr %0" : : "r" (sel));
}

static inline uint
readeflags(void)
{
  uint eflags;
  asm volatile("pushfl; popl %0" : "=r" (eflags));
  return eflags;
}
```

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}

static inline void
loadgs(ushort v)
{
  asm volatile("movw %0, %%gs" : : "r" (v));
}

static inline void
cli(void)
{
  asm volatile("cli");
}

static inline void
sti(void)
{
  asm volatile("sti");
}
```

```
// Layout of the trap frame built on the stack by the
// hardware and by trapasm.S, and passed to trap().
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding6;
};
```

```
//
// assembler macros to create x86 segments
//

#define SEG_NULLASM                                             \
        .word 0, 0;                                             \
        .byte 0, 0, 0, 0

#define SEG_ASM(type,base,lim)                                  \
        .word (((lim) >> 12) & 0xffff), ((base) & 0xffff);      \
        .byte (((base) >> 16) & 0xff), (0x90 | (type)),         \
                (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)

#define STA_X     0x8       // Executable segment
#define STA_E     0x4       // Expand down (non-executable segments)
#define STA_C     0x4       // Conforming code segment (executable only)
#define STA_W     0x2       // Writeable (non-executable segments)
#define STA_R     0x2       // Readable (executable segments)
#define STA_A     0x1       // Accessed
```

```
// This file contains definitions for the
// x86 memory management unit (MMU).

// Eflags register
#define FL_CF           0x00000001      // Carry Flag
#define FL_PF           0x00000004      // Parity Flag
#define FL_AF           0x00000010      // Auxiliary carry Flag
#define FL_ZF           0x00000040      // Zero Flag
#define FL_SF           0x00000080      // Sign Flag
#define FL_TF           0x00000100      // Trap Flag
#define FL_IF           0x00000200      // Interrupt Enable
#define FL_DF           0x00000400      // Direction Flag
#define FL_OF           0x00000800      // Overflow Flag
#define FL_IOPL_MASK    0x00003000      // I/O Privilege Level bitmask
#define FL_IOPL_0       0x00000000      //   IOPL == 0
#define FL_IOPL_1       0x00001000      //   IOPL == 1
#define FL_IOPL_2       0x00002000      //   IOPL == 2
#define FL_IOPL_3       0x00003000      //   IOPL == 3
#define FL_NT           0x00004000      // Nested Task
#define FL_RF           0x00010000      // Resume Flag
#define FL_VM           0x00020000      // Virtual 8086 mode
#define FL_AC           0x00040000      // Alignment Check
#define FL_VIF          0x00080000      // Virtual Interrupt Flag
#define FL_VIP          0x00100000      // Virtual Interrupt Pending
#define FL_ID           0x00200000      // ID flag

// Segment Descriptor
struct segdesc {
  uint lim_15_0 : 16;  // Low bits of segment limit
  uint base_15_0 : 16; // Low bits of segment base address
  uint base_23_16 : 8; // Middle bits of segment base address
  uint type : 4;       // Segment type (see STS_ constants)
  uint s : 1;          // 0 = system, 1 = application
  uint dpl : 2;        // Descriptor Privilege Level
  uint p : 1;          // Present
  uint lim_19_16 : 4;  // High bits of segment limit
  uint avl : 1;        // Unused (available for software use)
  uint rsv1 : 1;       // Reserved
  uint db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
  uint g : 1;          // Granularity: limit scaled by 4K when set
  uint base_31_24 : 8; // High bits of segment base address
};
```

```
// Normal segment
#define SEG(type, base, lim, dpl) (struct segdesc)    \
{ ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff,      \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
  (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }

#define SEG16(type, base, lim, dpl) (struct segdesc)  \
{ (lim) & 0xffff, (uint)(base) & 0xffff,              \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
  (uint)(lim) >> 16, 0, 0, 1, 0, (uint)(base) >> 24 }

#define DPL_USER    0x3     // User DPL

// Application segment type bits
#define STA_X       0x8     // Executable segment
#define STA_E       0x4     // Expand down (non-executable segments)
#define STA_C       0x4     // Conforming code segment (executable only)
#define STA_W       0x2     // Writeable (non-executable segments)
#define STA_R       0x2     // Readable (executable segments)
#define STA_A       0x1     // Accessed

// System segment type bits
#define STS_T16A    0x1     // Available 16-bit TSS
#define STS_LDT     0x2     // Local Descriptor Table
#define STS_T16B    0x3     // Busy 16-bit TSS
#define STS_CG16    0x4     // 16-bit Call Gate
#define STS_TG      0x5     // Task Gate / Coum Transmitions
#define STS_IG16    0x6     // 16-bit Interrupt Gate
#define STS_TG16    0x7     // 16-bit Trap Gate
#define STS_T32A    0x9     // Available 32-bit TSS
#define STS_T32B    0xB     // Busy 32-bit TSS
#define STS_CG32    0xC     // 32-bit Call Gate
#define STS_IG32    0xE     // 32-bit Interrupt Gate
#define STS_TG32    0xF     // 32-bit Trap Gate
```

```
// Task state segment format
struct taskstate {
  uint link;         // Old ts selector
  uint esp0;         // Stack pointers and segment selectors
  ushort ss0;        //   after an increase in privilege level
  ushort padding1;
  uint *esp1;
  ushort ss1;
  ushort padding2;
  uint *esp2;
  ushort ss2;
  ushort padding3;
  void *cr3;         // Page directory base
  uint *eip;         // Saved state from last task switch
  uint eflags;
  uint eax;          // More saved state (registers)
  uint ecx;
  uint edx;
  uint ebx;
  uint *esp;
  uint *ebp;
  uint esi;
  uint edi;
  ushort es;         // Even more saved state (segment selectors)
  ushort padding4;
  ushort cs;
  ushort padding5;
  ushort ss;
  ushort padding6;
  ushort ds;
  ushort padding7;
  ushort fs;
  ushort padding8;
  ushort gs;
  ushort padding9;
  ushort ldt;
  ushort padding10;
  ushort t;          // Trap on task switch
  ushort iomb;       // I/O map base address
};
```

```
// Gate descriptors for interrupts and traps
struct gatedesc {
  uint off_15_0 : 16;   // low 16 bits of offset in segment
  uint cs : 16;         // code segment selector
  uint args : 5;        // # args, 0 for interrupt/trap gates
  uint rsv1 : 3;        // reserved(should be zero I guess)
  uint type : 4;        // type(STS_{TG,IG32,TG32})
  uint s : 1;           // must be 0 (system)
  uint dpl : 2;         // descriptor(meaning new) privilege level
  uint p : 1;           // Present
  uint off_31_16 : 16;  // high bits of offset in segment
};

// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
//   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//        the privilege level required for software to invoke
//        this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, d)                \
{                                                         \
  (gate).off_15_0 = (uint) (off) & 0xffff;                \
  (gate).cs = (sel);                                      \
  (gate).args = 0;                                        \
  (gate).rsv1 = 0;                                        \
  (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
  (gate).s = 0;                                           \
  (gate).dpl = (d);                                       \
  (gate).p = 1;                                           \
  (gate).off_31_16 = (uint) (off) >> 16;                  \
}
```

```
// Format of an ELF executable file

#define ELF_MAGIC 0x464C457FU  // "\x7FELF" in little endian

// File header
struct elfhdr {
  uint magic;  // must equal ELF_MAGIC
  uchar elf[12];
  ushort type;
  ushort machine;
  uint version;
  uint entry;
  uint phoff;
  uint shoff;
  uint flags;
  ushort ehsize;
  ushort phentsize;
  ushort phnum;
  ushort shentsize;
  ushort shnum;
  ushort shstrndx;
};

// Program section header
struct proghdr {
  uint type;
  uint offset;
  uint va;
  uint pa;
  uint filesz;
  uint memsz;
  uint flags;
  uint align;
};

// Values for Proghdr type
#define ELF_PROG_LOAD           1

// Flag bits for Proghdr flags
#define ELF_PROG_FLAG_EXEC      1
#define ELF_PROG_FLAG_WRITE     2
#define ELF_PROG_FLAG_READ      4
```

```
#include "asm.h"

# Start the first CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

#define SEG_KCODE 1  // kernel code
#define SEG_KDATA 2  // kernel data+stack

#define CR0_PE    1  // protected mode enable bit

.code16                       # Assemble for 16-bit mode
.globl start
start:
  cli                         # Disable interrupts

  # Set up the important data segment registers (DS, ES, SS).
  xorw    %ax,%ax             # Segment number zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment

  # Enable A20:
  #   For backwards compatibility with the earliest PCs, physical
  #   address line 20 is tied low, so that addresses higher than
  #   1MB wrap around to zero by default.  This code undoes this.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al               # 0xdf -> port 0x60
  outb    %al,$0x60
```

```
  # Switch from real to protected mode, using a bootstrap GDT
  # and segment translation that makes virtual addresses
  # identical to physical addresses, so that the
  # effective memory map does not change during the switch.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE, %eax
  movl    %eax, %cr0

  # Jump to next instruction, but in 32-bit code segment.
  # Switches processor into 32-bit mode.
  ljmp    $(SEG_KCODE<<3), $start32

.code32                       # Assemble for 32-bit mode
start32:
  # Set up the protected-mode data segment registers
  movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %ss                # -> SS: Stack Segment
  movw    $0, %ax                 # Zero segments not ready for use
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS

  # Set up the stack pointer and call into C.
  movl    $start, %esp
  call    bootmain

  # If bootmain returns (it shouldn't), trigger a Bochs
  # breakpoint if running under Bochs, then loop.
  movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
  movw    %ax, %dx
  outw    %ax, %dx
  movw    $0x8e00, %ax            # 0x8e00 -> port 0x8a00
  outw    %ax, %dx
spin:
  jmp     spin

# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
  SEG_NULLASM                             # null seg
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
  SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg

gdtdesc:
  .word   (gdtdesc - gdt - 1)                            # sizeof(gdt) - 1
  .long   gdt                             # address gdt
```

```
#include "asm.h"

# Each non-boot CPU ("AP") is started up in response to a STARTUP
# IPI from the boot CPU.  Section B.4.2 of the Multi-Processor
# Specification says that the AP will start in real mode with CS:IP
# set to XY00:0000, where XY is an 8-bit value sent with the
# STARTUP. Thus this code must start at a 4096-byte boundary.
#
# Because this code sets DS to zero, it must sit
# at an address in the low 2^16 bytes.
#
# Bootothers (in main.c) sends the STARTUPs, one at a time.
# It puts this code (start) at 0x7000.
# It puts the correct %esp in start-4,
# and the place to jump to in start-8.
#
# This code is identical to bootasm.S except:
#   - it does not need to enable A20
#   - it uses the address at start-4 for the %esp
#   - it jumps to the address at start-8 instead of calling bootmain

#define SEG_KCODE 1  // kernel code
#define SEG_KDATA 2  // kernel data+stack

#define CR0_PE    1  // protected mode enable bit

.code16                       # Assemble for 16-bit mode
.globl start
start:
  cli                         # Disable interrupts

  # Set up the important data segment registers (DS, ES, SS).
  xorw    %ax,%ax             # Segment number zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment
```

```
  # Switch from real to protected mode, using a bootstrap GDT
  # and segment translation that makes virtual addresses
  # identical to physical addresses, so that the
  # effective memory map does not change during the switch.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE, %eax
  movl    %eax, %cr0

  # Jump to next instruction, but in 32-bit code segment.
  # Switches processor into 32-bit mode.
  ljmp    $(SEG_KCODE<<3), $start32

.code32                       # Assemble for 32-bit mode
start32:
  # Set up the protected-mode data segment registers
  movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %ss                # -> SS: Stack Segment
  movw    $0, %ax                 # Zero segments not ready for use
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS

  # Set up the stack pointer and call into C.
  movl    start-4, %esp
  call     *(start-8)

  # If the call returns (it shouldn't), trigger a Bochs
  # breakpoint if running under Bochs, then loop.
  movw    $0x8a00, %ax            # 0x8a00 -> port 0x8a00
  movw    %ax, %dx
  outw    %ax, %dx
  movw    $0x8e00, %ax            # 0x8e00 -> port 0x8a00
  outw    %ax, %dx
spin:
  jmp     spin

# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
  SEG_NULLASM                             # null seg
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
  SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg

gdtdesc:
  .word   (gdtdesc - gdt - 1)                             # sizeof(gdt) - 1
  .long   gdt                             # address gdt
```

```c
// Boot loader.
//
// Part of the boot sector, along with bootasm.S, which calls bootmain().
// bootasm.S has put the processor into protected 32-bit mode.
// bootmain() loads an ELF kernel image from the disk starting at
// sector 1 and then jumps to the kernel entry routine.

#include "types.h"
#include "elf.h"
#include "x86.h"

#define SECTSIZE  512

void readseg(uchar*, uint, uint);

void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* va;

  elf = (struct elfhdr*)0x10000;  // scratch space

  // Read 1st page off disk
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++) {
    va = (uchar*)(ph->va & 0xFFFFFF);
    readseg(va, ph->filesz, ph->offset);
    if(ph->memsz > ph->filesz)
      stosb(va + ph->filesz, 0, ph->memsz - ph->filesz);
  }

  // Call the entry point from the ELF header.
  // Does not return!
  entry = (void(*)(void))(elf->entry & 0xFFFFFF);
  entry();
}
```

```c
void
waitdisk(void)
{
  // Wait for disk ready.
  while((inb(0x1F7) & 0xC0) != 0x40)
    ;
}

// Read a single sector at offset into dst.
void
readsect(void *dst, uint offset)
{
  // Issue command.
  waitdisk();
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
  outb(0x1F5, offset >> 16);
  outb(0x1F6, (offset >> 24) | 0xE0);
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
  insl(0x1F0, dst, SECTSIZE/4);
}

// Read ’count’ bytes at ’offset’ from kernel into virtual address ’va’.
// Might copy more than asked.
void
readseg(uchar* va, uint count, uint offset)
{
  uchar* eva;

  eva = va + count;

  // Round down to sector boundary.
  va -= offset % SECTSIZE;

  // Translate from bytes to sectors; kernel starts at sector 1.
  offset = (offset / SECTSIZE) + 1;

  // If this is too slow, we could read lots of sectors at a time.
  // We’d write more to memory than asked, but it doesn’t matter --
  // we load in increasing order.
  for(; va < eva; va += SECTSIZE, offset++)
    readsect(va, offset);
}
```

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"

static void bootothers(void);
static void mpmain(void) __attribute__((noreturn));

// Bootstrap processor starts running C code here.
int
main(void)
{
  mpinit(); // collect info about this machine
  lapicinit(mpbcpu());
  ksegment();
  picinit();       // interrupt controller
  ioapicinit();    // another interrupt controller
  consoleinit();   // I/O devices & their interrupts
  uartinit();      // serial port
cprintf("cpus %p cpu %p\n", cpus, cpu);
  cprintf("\ncpu%d: starting xv6\n\n", cpu->id);

  kinit();         // physical memory allocator
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  iinit();         // inode cache
  ideinit();       // disk
  if(!ismp)
    timerinit();   // uniprocessor timer
  userinit();      // first user process
  bootothers();    // start other processors

  // Finish setting up this processor in mpmain.
  mpmain();
}
```

```
// Bootstrap processor gets here after setting up the hardware.
// Additional processors start here.
static void
mpmain(void)
{
  if(cpunum() != mpbcpu())
    lapicinit(cpunum());
  ksegment();
  cprintf("cpu%d: mpmain\n", cpu->id);
  idtinit();
  xchg(&cpu->booted, 1);

  cprintf("cpu%d: scheduling\n", cpu->id);
  scheduler();
}

static void
bootothers(void)
{
  extern uchar _binary_bootother_start[], _binary_bootother_size[];
  uchar *code;
  struct cpu *c;
  char *stack;

  // Write bootstrap code to unused memory at 0x7000.
  code = (uchar*)0x7000;
  memmove(code, _binary_bootother_start, (uint)_binary_bootother_size);

  for(c = cpus; c < cpus+ncpu; c++){
    if(c == cpus+cpunum())  // We’ve started already.
      continue;

    // Fill in %esp, %eip and start code on cpu.
    stack = kalloc(KSTACKSIZE);
    *(void**)(code-4) = stack + KSTACKSIZE;
    *(void**)(code-8) = mpmain;
    lapicstartap(c->id, (uint)code);

    // Wait for cpu to get through bootstrap.
    while(c->booted == 0)
      ;
  }
}
```

```
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

```
// Mutual exclusion spin locks.

#include "types.h"
#include "defs.h"
#include "param.h"
#include "x86.h"
#include "mmu.h"
#include "proc.h"
#include "spinlock.h"

void
initlock(struct spinlock *lk, char *name)
{
  lk->name = name;
  lk->locked = 0;
  lk->cpu = 0;
}

// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void
acquire(struct spinlock *lk)
{
  pushcli();
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  // It also serializes, so that reads after acquire are not
  // reordered before it.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Record info about lock acquisition for debugging.
  lk->cpu = cpu;
  getcallerpcs(&lk, lk->pcs);
}
```

```
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  // The xchg serializes, so that reads before release are
  // not reordered after it.  The 1996 PentiumPro manual (Volume 3,
  // 7.2) says reads can be carried out speculatively and in
  // any order, which implies we need to serialize here.
  // But the 2007 Intel 64 Architecture Memory Ordering White
  // Paper says that Intel 64 and IA-32 will not move a load
  // after a store. So lock->locked = 0 would work here.
  // The xchg being asm volatile ensures gcc emits it after
  // the above assignments (and after the critical section).
  xchg(&lk->locked, 0);

  popcli();
}

// Record the current call stack in pcs[] by following the %ebp chain.
void
getcallerpcs(void *v, uint pcs[])
{
  uint *ebp;
  int i;

  ebp = (uint*)v - 2;
  for(i = 0; i < 10; i++){
    if(ebp == 0 || ebp == (uint*)0xffffffff)
      break;
    pcs[i] = ebp[1];     // saved %eip
    ebp = (uint*)ebp[0]; // saved %ebp
  }
  for(; i < 10; i++)
    pcs[i] = 0;
}

// Check whether this cpu is holding the lock.
int
holding(struct spinlock *lock)
{
  return lock->locked && lock->cpu == cpu;
}
```

```
// Pushcli/popcli are like cli/sti except that they are matched:
// it takes two popcli to undo two pushcli.  Also, if interrupts
// are off, then pushcli, popcli leaves them off.

void
pushcli(void)
{
  int eflags;

  eflags = readeflags();
  cli();
  if(cpu->ncli++ == 0)
    cpu->intena = eflags & FL_IF;
}

void
popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--cpu->ncli < 0)
    panic("popcli");
  if(cpu->ncli == 0 && cpu->intena)
    sti();
}
```

```
// Segments in proc->gdt.
// Also known to bootasm.S and trapasm.S
#define SEG_KCODE 1  // kernel code
#define SEG_KDATA 2  // kernel data+stack
#define SEG_KCPU  3  // kernel per-cpu data
#define SEG_UCODE 4
#define SEG_UDATA 5
#define SEG_TSS   6  // this process’s task state
#define NSEGS     7

// Saved registers for kernel context switches.
// Don’t need to save all the segment registers (%cs, etc),
// because they are constant across kernel contexts.
// Don’t need to save %eax, %ecx, %edx, because the
// x86 convention is that the caller has saved them.
// Contexts are stored at the bottom of the stack they
// describe; the stack pointer is the address of the context.
// The layout of the context must match the code in swtch.S.
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  char *mem;                    // Start of process memory (kernel address)
  uint sz;                      // Size of process memory (bytes)
  char *kstack;                 // Bottom of kernel stack for this process
  enum procstate state;         // Process state
  volatile int pid;             // Process ID
  struct proc *parent;          // Parent process
  struct trapframe *tf;         // Trap frame for current syscall
  struct context *context;      // Switch here to run process
  void *chan;                   // If non-zero, sleeping on chan
  int killed;                   // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;            // Current directory
  char name[16];                // Process name (debugging)
};
```

```
// Process memory is laid out contiguously, low addresses first:
//   text
//   original data and bss
//   fixed-size stack
//   expandable heap

// Per-CPU state
struct cpu {
  uchar id;                    // Local APIC ID; index into cpus[] below
  struct context *scheduler;   // Switch here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint booted;        // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?

  // Cpu-local storage variables; see below
  struct cpu *cpu;
  struct proc *proc;
};

extern struct cpu cpus[NCPU];
extern int ncpu;

// Per-CPU variables, holding pointers to the
// current cpu and to the current process.
// The asm suffix tells gcc to use "%gs:0" to refer to cpu
// and "%gs:4" to refer to proc.  ksegment sets up the
// %gs segment register so that %gs refers to the memory
// holding those two variables in the local cpu’s struct cpu.
// This is similar to how thread-local variables are implemented
// in thread libraries such as Linux pthreads.
extern struct cpu *cpu asm("%gs:0");       // This cpu.
extern struct proc *proc asm("%gs:4");     // Current proc on this cpu.
```

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "mmu.h"
#include "x86.h"
#include "proc.h"
#include "spinlock.h"

struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;

static struct proc *initproc;

int nextpid = 1;
extern void forkret(void);
extern void trapret(void);

void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
}
```

```
// Print a process listing to console.  For debugging.
// Runs when user types ^P on console.
// No lock to avoid wedging a stuck machine further.
void
procdump(void)
{
  static char *states[] = {
  [UNUSED]    "unused",
  [EMBRYO]    "embryo",
  [SLEEPING]  "sleep ",
  [RUNNABLE]  "runble",
  [RUNNING]   "run   ",
  [ZOMBIE]    "zombie"
  };
  int i;
  struct proc *p;
  char *state;
  uint pc[10];

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == UNUSED)
      continue;
    if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
      state = states[p->state];
    else
      state = "???";
    cprintf("%d %s %s", p->pid, state, p->name);
    if(p->state == SLEEPING){
      getcallerpcs((uint*)p->context->ebp+2, pc);
      for(i=0; i<10 && pc[i] != 0; i++)
        cprintf(" %p", pc[i]);
    }
    cprintf("\n");
  }
}
```

```
// Set up CPU’s kernel segment descriptors.
// Run once at boot time on each CPU.
void
ksegment(void)
{
  struct cpu *c;

  c = &cpus[cpunum()];
  c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0x100000 + 64*1024-1, 0);
  c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
  c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
  lgdt(c->gdt, sizeof(c->gdt));
  loadgs(SEG_KCPU << 3);

  // Initialize cpu-local storage.
  cpu = c;
  proc = 0;
}

// Set up CPU’s segment descriptors and current process task state.
void
usegment(void)
{
  pushcli();
  cpu->gdt[SEG_UCODE] = SEG(STA_X|STA_R, proc->mem, proc->sz-1, DPL_USER);
  cpu->gdt[SEG_UDATA] = SEG(STA_W, proc->mem, proc->sz-1, DPL_USER);
  cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
  cpu->gdt[SEG_TSS].s = 0;
  cpu->ts.ss0 = SEG_KDATA << 3;
  cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
  ltr(SEG_TSS << 3);
  popcli();
}
```

```
// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and return it.
// Otherwise return 0.
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;
  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  release(&ptable.lock);

  // Allocate kernel stack if necessary.
  if((p->kstack = kalloc(KSTACKSIZE)) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret (see below).
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  return p;
}
```

```
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();
  initproc = p;

  // Initialize memory from initcode.S
  p->sz = PAGE;
  p->mem = kalloc(p->sz);
  memset(p->mem, 0, p->sz);
  memmove(p->mem, _binary_initcode_start, (int)_binary_initcode_size);

  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = p->sz;
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  p->state = RUNNABLE;
}

// Grow current process's memory by n bytes.
// Return 0 on success, -1 on failure.
int
growproc(int n)
{
  char *newmem;

  newmem = kalloc(proc->sz + n);
  if(newmem == 0)
    return -1;
  memmove(newmem, proc->mem, proc->sz);
  memset(newmem + proc->sz, 0, n);
  kfree(proc->mem, proc->sz);
  proc->mem = newmem;
  proc->sz += n;
  usegment();
  return 0;
}
```

```
// Create a new process copying p as the parent.
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.
int
fork(void)
{
  int i, pid;
  struct proc *np;

  // Allocate process.
  if((np = allocproc()) == 0)
    return -1;

  // Copy process state from p.
  np->sz = proc->sz;
  if((np->mem = kalloc(np->sz)) == 0){
    kfree(np->kstack, KSTACKSIZE);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  memmove(np->mem, proc->mem, np->sz);
  np->parent = proc;
  *np->tf = *proc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(proc->ofile[i])
      np->ofile[i] = filedup(proc->ofile[i]);
  np->cwd = idup(proc->cwd);

  pid = np->pid;
  np->state = RUNNABLE;

  return pid;
}
```

```
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      usegment();
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);

  }
}
```

```
// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state.
void
sched(void)
{
  int intena;

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(cpu->ncli != 1)
    panic("sched locks");
  if(proc->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");

  intena = cpu->intena;
  swtch(&proc->context, cpu->scheduler);
  cpu->intena = intena;
}

// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);
  proc->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}

// A fork child's very first scheduling by scheduler()
// will swtch here.  "Return" to user space.
void
forkret(void)
{
  // Still holding ptable.lock from scheduler.
  release(&ptable.lock);

  // Return to "caller", actually trapret (see allocproc).
}
```

```
// Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
void
sleep(void *chan, struct spinlock *lk)
{
  if(proc == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won’t miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it’s okay to release lk.
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }

  // Go to sleep.
  proc->chan = chan;
  proc->state = SLEEPING;
  sched();

  // Tidy up.
  proc->chan = 0;

  // Reacquire original lock.
  if(lk != &ptable.lock){
    release(&ptable.lock);
    acquire(lk);
  }
}
```

```
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}

// Wake up all processes sleeping on chan.
void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}

// Kill the process with the given pid.
// Process won’t exit until it returns
// to user space (see trap in trap.c).
int
kill(int pid)
{
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->killed = 1;
      // Wake process from sleep if necessary.
      if(p->state == SLEEPING)
        p->state = RUNNABLE;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

```
// Exit the current process.  Does not return.
// An exited process remains in the zombie state
// until its parent calls wait() to find out it exited.
void
exit(void)
{
  struct proc *p;
  int fd;

  if(proc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(proc->ofile[fd]){
      fileclose(proc->ofile[fd]);
      proc->ofile[fd] = 0;
    }
  }

  iput(proc->cwd);
  proc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(proc->parent);

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == proc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  proc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

```
// Wait for a child process to exit and return its pid.
// Return -1 if this process has no children.
int
wait(void)
{
  struct proc *p;
  int havekids, pid;

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for zombie children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != proc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->mem, p->sz);
        kfree(p->kstack, KSTACKSIZE);
        p->state = UNUSED;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        release(&ptable.lock);
        return pid;
      }
    }

    // No point waiting if we don’t have any children.
    if(!havekids || proc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(proc, &ptable.lock);
  }
}
```

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

```
// Physical memory allocator, intended to allocate
// memory for user processes. Allocates in 4096-byte "pages".
// Free list is kept sorted and combines adjacent pages into
// long runs, to make it easier to allocate big segments.
// One reason the page size is 4k is that the x86 segment size
// granularity is 4k.

#include "types.h"
#include "defs.h"
#include "param.h"
#include "spinlock.h"

struct run {
  struct run *next;
  int len; // bytes
};

struct {
  struct spinlock lock;
  struct run *freelist;
} kmem;

// Initialize free list of physical pages.
// This code cheats by just considering one megabyte of
// pages after end.  Real systems would determine the
// amount of memory available in the system and use it all.
void
kinit(void)
{
  extern char end[];
  uint len;
  char *p;

  initlock(&kmem.lock, "kmem");
  p = (char*)(((uint)end + PAGE) & ~(PAGE-1));
  len = 256*PAGE; // assume computer has 256 pages of RAM, 1 MB
  cprintf("mem = %d\n", len);
  kfree(p, len);
}
```

```
// Free the len bytes of memory pointed at by v,
// which normally should have been returned by a
// call to kalloc(len).  (The exception is when
// initializing the allocator; see kinit above.)
void
kfree(char *v, int len)
{
  struct run *r, *rend, **rp, *p, *pend;

  if(len <= 0 || len % PAGE)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(v, 1, len);

  acquire(&kmem.lock);
  p = (struct run*)v;
  pend = (struct run*)(v + len);
  for(rp=&kmem.freelist; (r=*rp) != 0 && r <= pend; rp=&r->next){
    rend = (struct run*)((char*)r + r->len);
    if(r <= p && p < rend)
      panic("freeing free page");
    if(rend == p){  // r before p: expand r to include p
      r->len += len;
      if(r->next && r->next == pend){  // r now next to r->next?
        r->len += r->next->len;
        r->next = r->next->next;
      }
      goto out;
    }
    if(pend == r){  // p before r: expand p to include, replace r
      p->len = len + r->len;
      p->next = r->next;
      *rp = p;
      goto out;
    }
  }
  // Insert p before r in list.
  p->len = len;
  p->next = r;
  *rp = p;

 out:
  release(&kmem.lock);
}
```

```
// Allocate n bytes of physical memory.
// Returns a kernel-segment pointer.
// Returns 0 if the memory cannot be allocated.
char*
kalloc(int n)
{
  char *p;
  struct run *r, **rp;

  if(n % PAGE || n <= 0)
    panic("kalloc");

  acquire(&kmem.lock);
  for(rp=&kmem.freelist; (r=*rp) != 0; rp=&r->next){
    if(r->len >= n){
      r->len -= n;
      p = (char*)r + r->len;
      if(r->len == 0)
        *rp = r->next;
      release(&kmem.lock);
      return p;
    }
  }
  release(&kmem.lock);

  cprintf("kalloc: out of memory\n");
  return 0;
}
```

```
// x86 trap and interrupt constants.

// Processor-defined:
#define T_DIVIDE         0      // divide error
#define T_DEBUG          1      // debug exception
#define T_NMI            2      // non-maskable interrupt
#define T_BRKPT          3      // breakpoint
#define T_OFLOW          4      // overflow
#define T_BOUND          5      // bounds check
#define T_ILLOP          6      // illegal opcode
#define T_DEVICE         7      // device not available
#define T_DBLFLT         8      // double fault
// #define T_COPROC      9      // reserved (not used since 486)
#define T_TSS           10      // invalid task switch segment
#define T_SEGNP         11      // segment not present
#define T_STACK         12      // stack exception
#define T_GPFLT         13      // general protection fault
#define T_PGFLT         14      // page fault
// #define T_RES        15      // reserved
#define T_FPERR         16      // floating point error
#define T_ALIGN         17      // aligment check
#define T_MCHK          18      // machine check
#define T_SIMDERR       19      // SIMD floating point error

// These are arbitrarily chosen, but with care not to overlap
// processor defined exceptions or interrupt vectors.
#define T_SYSCALL       64     // system call
#define T_DEFAULT      500      // catchall

#define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ

#define IRQ_TIMER        0
#define IRQ_KBD          1
#define IRQ_COM1         4
#define IRQ_IDE         14
#define IRQ_ERROR       19
#define IRQ_SPURIOUS    31
```

```
#!/usr/bin/perl -w

# Generate vectors.S, the trap/interrupt entry points.
# There has to be one entry point per interrupt number
# since otherwise there’s no way for trap() to discover
# the interrupt number.

print "# generated by vectors.pl - do not edit\n";
print "# handlers\n";
print ".globl alltraps\n";
for(my $i = 0; $i < 256; $i++){
    print ".globl vector$i\n";
    print "vector$i:\n";
    if(!($i == 8 || ($i >= 10 && $i <= 14) || $i == 17)){
        print "  pushl \$0\n";
    }
    print "  pushl \$$i\n";
    print "  jmp alltraps\n";
}

print "\n# vector table\n";
print ".data\n";
print ".globl vectors\n";
print "vectors:\n";
for(my $i = 0; $i < 256; $i++){
    print "  .long vector$i\n";
}

# sample output:
#   # handlers
#   .globl alltraps
#   .globl vector0
#   vector0:
#     pushl $0
#     pushl $0
#     jmp alltraps
#   ...
#
#   # vector table
#   .data
#   .globl vectors
#   vectors:
#     .long vector0
#     .long vector1
#     .long vector2
#   ...
```

```
#define SEG_KCODE 1  // kernel code
#define SEG_KDATA 2  // kernel data+stack
#define SEG_KCPU  3  // kernel per-cpu data

  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data and per-cpu segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es
  movw $(SEG_KCPU<<3), %ax
  movw %ax, %fs
  movw %ax, %gs

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "traps.h"
#include "spinlock.h"

// Interrupt descriptor table (shared by all CPUs).
struct gatedesc idt[256];
extern uint vectors[];  // in vectors.S: array of 256 entry pointers
struct spinlock tickslock;
int ticks;

void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}

void
idtinit(void)
{
  lidt(idt, sizeof(idt));
}
```

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(proc->killed)
      exit();
    proc->tf = tf;
    syscall();
    if(proc->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpu->id == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE:
    ideintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_COM1:
    uartintr();
    lapiceoi();
    break;
  case T_IRQ0 + 7:
  case T_IRQ0 + IRQ_SPURIOUS:
    cprintf("cpu%d: spurious interrupt at %x:%x\n",
            cpu->id, tf->cs, tf->eip);
    lapiceoi();
    break;
```

```
  default:
    if(proc == 0 || (tf->cs&3) == 0){
      // In kernel, it must be our mistake.
      cprintf("unexpected trap %d from cpu %d eip %x\n",
              tf->trapno, cpu->id, tf->eip);
      panic("trap");
    }
    // In user space, assume process misbehaved.
    cprintf("pid %d %s: trap %d err %d on cpu %d eip %x -- kill proc\n",
            proc->pid, proc->name, tf->trapno, tf->err, cpu->id, tf->eip);
    proc->killed = 1;
  }

  // Force process exit if it has been killed and is in user space.
  // (If it is still executing in the kernel, let it keep running
  // until it gets to the regular system call return.)
  if(proc && proc->killed && (tf->cs&3) == DPL_USER)
    exit();

  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
  if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();

  // Check if the process has been killed since we yielded
  if(proc && proc->killed && (tf->cs&3) == DPL_USER)
    exit();
}
```

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_write   5
#define SYS_read    6
#define SYS_close   7
#define SYS_kill    8
#define SYS_exec    9
#define SYS_open   10
#define SYS_mknod  11
#define SYS_unlink 12
#define SYS_fstat  13
#define SYS_link   14
#define SYS_mkdir  15
#define SYS_chdir  16
#define SYS_dup    17
#define SYS_getpid 18
#define SYS_sbrk   19
#define SYS_sleep  20
```

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "syscall.h"

// User code makes a system call with INT T_SYSCALL.
// System call number in %eax.
// Arguments on the stack, from the user call to the C
// library system call function. The saved user %esp points
// to a saved program counter, and then the first argument.

// Fetch the int at addr from process p.
int
fetchint(struct proc *p, uint addr, int *ip)
{
  if(addr >= p->sz || addr+4 > p->sz)
    return -1;
  *ip = *(int*)(p->mem + addr);
  return 0;
}

// Fetch the nul-terminated string at addr from process p.
// Doesn’t actually copy the string - just sets *pp to point at it.
// Returns length of string, not including nul.
int
fetchstr(struct proc *p, uint addr, char **pp)
{
  char *s, *ep;

  if(addr >= p->sz)
    return -1;
  *pp = p->mem + addr;
  ep = p->mem + p->sz;
  for(s = *pp; s < ep; s++)
    if(*s == 0)
      return s - *pp;
  return -1;
}

// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
  return fetchint(proc, proc->tf->esp + 4 + 4*n, ip);
}
```

```
// Fetch the nth word-sized system call argument as a pointer
// to a block of memory of size n bytes.  Check that the pointer
// lies within the process address space.
int
argptr(int n, char **pp, int size)
{
  int i;

  if(argint(n, &i) < 0)
    return -1;
  if((uint)i >= proc->sz || (uint)i+size >= proc->sz)
    return -1;
  *pp = proc->mem + i;
  return 0;
}

// Fetch the nth word-sized system call argument as a string pointer.
// Check that the pointer is valid and the string is nul-terminated.
// (There is no shared writable memory, so the string can't change
// between this check and being used by the kernel.)
int
argstr(int n, char **pp)
{
  int addr;
  if(argint(n, &addr) < 0)
    return -1;
  return fetchstr(proc, addr, pp);
}

extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
```

```
static int (*syscalls[])(void) = {
[SYS_chdir]   sys_chdir,
[SYS_close]   sys_close,
[SYS_dup]     sys_dup,
[SYS_exec]    sys_exec,
[SYS_exit]    sys_exit,
[SYS_fork]    sys_fork,
[SYS_fstat]   sys_fstat,
[SYS_getpid]  sys_getpid,
[SYS_kill]    sys_kill,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_mknod]   sys_mknod,
[SYS_open]    sys_open,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_unlink]  sys_unlink,
[SYS_wait]    sys_wait,
[SYS_write]   sys_write,
};

void
syscall(void)
{
  int num;

  num = proc->tf->eax;
  if(num >= 0 && num < NELEM(syscalls) && syscalls[num])
    proc->tf->eax = syscalls[num]();
  else {
    cprintf("%d %s: unknown sys call %d\n",
            proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}
```

```c
#include "types.h"
#include "x86.h"
#include "defs.h"
#include "param.h"
#include "mmu.h"
#include "proc.h"

int
sys_fork(void)
{
  return fork();
}

int
sys_exit(void)
{
  exit();
  return 0;  // not reached
}

int
sys_wait(void)
{
  return wait();
}

int
sys_kill(void)
{
  int pid;

  if(argint(0, &pid) < 0)
    return -1;
  return kill(pid);
}

int
sys_getpid(void)
{
  return proc->pid;
}
```

```c
int
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;
  addr = proc->sz;
  if(growproc(n) < 0)
    return -1;
  return addr;
}

int
sys_sleep(void)
{
  int n, ticks0;

  if(argint(0, &n) < 0)
    return -1;
  acquire(&tickslock);
  ticks0 = ticks;
  while(ticks - ticks0 < n){
    if(proc->killed){
      release(&tickslock);
      return -1;
    }
    sleep(&ticks, &tickslock);
  }
  release(&tickslock);
  return 0;
}
```

```
struct buf {
  int flags;
  uint dev;
  uint sector;
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[512];
};
#define B_BUSY  0x1  // buffer is locked by some process
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk
```

```
#define O_RDONLY  0x000
#define O_WRONLY  0x001
#define O_RDWR    0x002
#define O_CREATE  0x200
```

```
#define T_DIR  1   // Directory
#define T_FILE 2   // File
#define T_DEV  3   // Special device

struct stat {
  short type;  // Type of file
  int dev;     // Device number
  uint ino;    // Inode number on device
  short nlink; // Number of links to file
  uint size;   // Size of file in bytes
};
```

```
// On-disk file system format.
// Both the kernel and user programs use this header file.

// Block 0 is unused.
// Block 1 is super block.
// Inodes start at block 2.

#define ROOTINO 1  // root i-number
#define BSIZE 512  // block size

// File system super block
struct superblock {
  uint size;         // Size of file system image (blocks)
  uint nblocks;      // Number of data blocks
  uint ninodes;      // Number of inodes.
};

#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT)

// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};

// Inodes per block.
#define IPB           (BSIZE / sizeof(struct dinode))

// Block containing inode i
#define IBLOCK(i)     ((i) / IPB + 2)

// Bitmap bits per block
#define BPB           (BSIZE*8)

// Block containing bit for block b
#define BBLOCK(b, ninodes) (b/BPB + (ninodes)/IPB + 3)
```

```
// Directory is a file containing a sequence of dirent structures.
#define DIRSIZ 14

struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};


// in-core file system types

struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  int flags;          // I_BUSY, I_VALID

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};

#define I_BUSY 0x1
#define I_VALID 0x2


// device implementations

struct devsw {
  int (*read)(struct inode*, char*, int);
  int (*write)(struct inode*, char*, int);
};

extern struct devsw devsw[];

#define CONSOLE 1
```

```c
// Simple PIO-based (non-DMA) IDE driver code.

#include "types.h"
#include "defs.h"
#include "param.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"
#include "traps.h"
#include "spinlock.h"
#include "buf.h"

#define IDE_BSY       0x80
#define IDE_DRDY      0x40
#define IDE_DF        0x20
#define IDE_ERR       0x01

#define IDE_CMD_READ  0x20
#define IDE_CMD_WRITE 0x30

// idequeue points to the buf now being read/written to the disk.
// idequeue->qnext points to the next buf to be processed.
// You must hold idelock while manipulating queue.

static struct spinlock idelock;
static struct buf *idequeue;

static int havedisk1;
static void idestart(struct buf*);

// Wait for IDE disk to become ready.
static int
idewait(int checkerr)
{
  int r;

  while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
    ;
  if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
    return -1;
  return 0;
}
```

```c
void
ideinit(void)
{
  int i;

  initlock(&idelock, "ide");
  picenable(IRQ_IDE);
  ioapicenable(IRQ_IDE, ncpu - 1);
  idewait(0);

  // Check if disk 1 is present
  outb(0x1f6, 0xe0 | (1<<4));
  for(i=0; i<1000; i++){
    if(inb(0x1f7) != 0){
      havedisk1 = 1;
      break;
    }
  }

  // Switch back to disk 0.
  outb(0x1f6, 0xe0 | (0<<4));
}

// Start the request for b.  Caller must hold idelock.
static void
idestart(struct buf *b)
{
  if(b == 0)
    panic("idestart");

  idewait(0);
  outb(0x3f6, 0);  // generate interrupt
  outb(0x1f2, 1);  // number of sectors
  outb(0x1f3, b->sector & 0xff);
  outb(0x1f4, (b->sector >> 8) & 0xff);
  outb(0x1f5, (b->sector >> 16) & 0xff);
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, IDE_CMD_WRITE);
    outsl(0x1f0, b->data, 512/4);
  } else {
    outb(0x1f7, IDE_CMD_READ);
  }
}
```

```c
// Interrupt handler.
void
ideintr(void)
{
  struct buf *b;

  // Take first buffer off queue.
  acquire(&idelock);
  if((b = idequeue) == 0){
    release(&idelock);
    cprintf("Spurious IDE interrupt.\n");
    return;
  }
  idequeue = b->qnext;

  // Read data if needed.
  if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
    insl(0x1f0, b->data, 512/4);

  // Wake process waiting for this buf.
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);

  // Start disk on next buf in queue.
  if(idequeue != 0)
    idestart(idequeue);

  release(&idelock);
}
```

```c
// Sync buf with disk.
// If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
// Else if B_VALID is not set, read buf from disk, set B_VALID.
void
iderw(struct buf *b)
{
  struct buf **pp;

  if(!(b->flags & B_BUSY))
    panic("iderw: buf not busy");
  if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
    panic("iderw: nothing to do");
  if(b->dev != 0 && !havedisk1)
    panic("idrw: ide disk 1 not present");

  acquire(&idelock);

  // Append b to idequeue.
  b->qnext = 0;
  for(pp=&idequeue; *pp; pp=&(*pp)->qnext)
    ;
  *pp = b;

  // Start disk if necessary.
  if(idequeue == b)
    idestart(b);

  // Wait for request to finish.
  // Assuming will not sleep too long: ignore proc->killed.
  while((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
    sleep(b, &idelock);

  release(&idelock);
}
```

```c
// Buffer cache.
//
// The buffer cache is a linked list of buf structures holding
// cached copies of disk block contents.  Caching disk blocks
// in memory reduces the number of disk reads and also provides
// a synchronization point for disk blocks used by multiple processes.
//
// Interface:
// * To get a buffer for a particular disk block, call bread.
// * After changing buffer data, call bwrite to flush it to disk.
// * When done with the buffer, call brelse.
// * Do not use the buffer after calling brelse.
// * Only one process at a time can use a buffer,
//     so do not keep them longer than necessary.
//
// The implementation uses three state flags internally:
// * B_BUSY: the block has been returned from bread
//     and has not been passed back to brelse.
// * B_VALID: the buffer data has been initialized
//     with the associated disk block contents.
// * B_DIRTY: the buffer data has been modified
//     and needs to be written to disk.

#include "types.h"
#include "defs.h"
#include "param.h"
#include "spinlock.h"
#include "buf.h"

struct {
  struct spinlock lock;
  struct buf buf[NBUF];

  // Linked list of all buffers, through prev/next.
  // head.next is most recently used.
  struct buf head;
} bcache;

void
binit(void)
{
  struct buf *b;

  initlock(&bcache.lock, "bcache");







  // Create linked list of buffers
  bcache.head.prev = &bcache.head;
  bcache.head.next = &bcache.head;
  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
    b->next = bcache.head.next;
    b->prev = &bcache.head;
    b->dev = -1;
    bcache.head.next->prev = b;
    bcache.head.next = b;
  }
}

// Look through buffer cache for sector on device dev.
// If not found, allocate fresh block.
// In either case, return locked buffer.
static struct buf*
bget(uint dev, uint sector)
{
  struct buf *b;

  acquire(&bcache.lock);

 loop:
  // Try for cached block.
  for(b = bcache.head.next; b != &bcache.head; b = b->next){
    if(b->dev == dev && b->sector == sector){
      if(!(b->flags & B_BUSY)){
        b->flags |= B_BUSY;
        release(&bcache.lock);
        return b;
      }
      sleep(b, &bcache.lock);
      goto loop;
    }
  }

  // Allocate fresh block.
  for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
    if((b->flags & B_BUSY) == 0){
      b->dev = dev;
      b->sector = sector;
      b->flags = B_BUSY;
      release(&bcache.lock);
      return b;
    }
  }
  panic("bget: no buffers");
}
```

```
// Return a B_BUSY buf with the contents of the indicated disk sector.
struct buf*
bread(uint dev, uint sector)
{
  struct buf *b;

  b = bget(dev, sector);
  if(!(b->flags & B_VALID))
    iderw(b);
  return b;
}

// Write b’s contents to disk.  Must be locked.
void
bwrite(struct buf *b)
{
  if((b->flags & B_BUSY) == 0)
    panic("bwrite");
  b->flags |= B_DIRTY;
  iderw(b);
}

// Release the buffer b.
void
brelse(struct buf *b)
{
  if((b->flags & B_BUSY) == 0)
    panic("brelse");

  acquire(&bcache.lock);

  b->next->prev = b->prev;
  b->prev->next = b->next;
  b->next = bcache.head.next;
  b->prev = &bcache.head;
  bcache.head.next->prev = b;
  bcache.head.next = b;

  b->flags &= ~B_BUSY;
  wakeup(b);

  release(&bcache.lock);
}
```

```
// File system implementation.  Four layers:
//   + Blocks: allocator for raw disk blocks.
//   + Files: inode allocator, reading, writing, metadata.
//   + Directories: inode with special contents (list of other inodes!)
//   + Names: paths like /usr/rtm/xv6/fs.c for convenient naming.
//
// Disk layout is: superblock, inodes, block in-use bitmap, data blocks.
//
// This file contains the low-level file system manipulation
// routines.  The (higher-level) system call implementations
// are in sysfile.c.

#include "types.h"
#include "defs.h"
#include "param.h"
#include "stat.h"
#include "mmu.h"
#include "proc.h"
#include "spinlock.h"
#include "buf.h"
#include "fs.h"
#include "file.h"

#define min(a, b) ((a) < (b) ? (a) : (b))
static void itrunc(struct inode*);

// Read the super block.
static void
readsb(int dev, struct superblock *sb)
{
  struct buf *bp;

  bp = bread(dev, 1);
  memmove(sb, bp->data, sizeof(*sb));
  brelse(bp);
}

// Zero a block.
static void
bzero(int dev, int bno)
{
  struct buf *bp;

  bp = bread(dev, bno);
  memset(bp->data, 0, BSIZE);
  bwrite(bp);
  brelse(bp);
}
```

```
// Blocks.

// Allocate a disk block.
static uint
balloc(uint dev)
{
  int b, bi, m;
  struct buf *bp;
  struct superblock sb;

  bp = 0;
  readsb(dev, &sb);
  for(b = 0; b < sb.size; b += BPB){
    bp = bread(dev, BBLOCK(b, sb.ninodes));
    for(bi = 0; bi < BPB; bi++){
      m = 1 << (bi % 8);
      if((bp->data[bi/8] & m) == 0){  // Is block free?
        bp->data[bi/8] |= m;  // Mark block in use on disk.
        bwrite(bp);
        brelse(bp);
        return b + bi;
      }
    }
    brelse(bp);
  }
  panic("balloc: out of blocks");
}

// Free a disk block.
static void
bfree(int dev, uint b)
{
  struct buf *bp;
  struct superblock sb;
  int bi, m;

  bzero(dev, b);

  readsb(dev, &sb);
  bp = bread(dev, BBLOCK(b, sb.ninodes));
  bi = b % BPB;
  m = 1 << (bi % 8);
  if((bp->data[bi/8] & m) == 0)
    panic("freeing free block");
  bp->data[bi/8] &= ~m;  // Mark block free on disk.
  bwrite(bp);
  brelse(bp);
}
```

```
// Inodes.
//
// An inode is a single, unnamed file in the file system.
// The inode disk structure holds metadata (the type, device numbers,
// and data size) along with a list of blocks where the associated
// data can be found.
//
// The inodes are laid out sequentially on disk immediately after
// the superblock.  The kernel keeps a cache of the in-use
// on-disk structures to provide a place for synchronizing access
// to inodes shared between multiple processes.
//
// ip->ref counts the number of pointer references to this cached
// inode; references are typically kept in struct file and in proc->cwd.
// When ip->ref falls to zero, the inode is no longer cached.
// It is an error to use an inode without holding a reference to it.
//
// Processes are only allowed to read and write inode
// metadata and contents when holding the inode's lock,
// represented by the I_BUSY flag in the in-memory copy.
// Because inode locks are held during disk accesses,
// they are implemented using a flag rather than with
// spin locks.  Callers are responsible for locking
// inodes before passing them to routines in this file; leaving
// this responsibility with the caller makes it possible for them
// to create arbitrarily-sized atomic operations.
//
// To give maximum control over locking to the callers,
// the routines in this file that return inode pointers
// return pointers to *unlocked* inodes.  It is the callers'
// responsibility to lock them before using them.  A non-zero
// ip->ref keeps these unlocked inodes in the cache.

struct {
  struct spinlock lock;
  struct inode inode[NINODE];
} icache;

void
iinit(void)
{
  initlock(&icache.lock, "icache");
}

static struct inode* iget(uint dev, uint inum);
```

```
// Allocate a new inode with the given type on device dev.
struct inode*
ialloc(uint dev, short type)
{
  int inum;
  struct buf *bp;
  struct dinode *dip;
  struct superblock sb;

  readsb(dev, &sb);
  for(inum = 1; inum < sb.ninodes; inum++){  // loop over inode blocks
    bp = bread(dev, IBLOCK(inum));
    dip = (struct dinode*)bp->data + inum%IPB;
    if(dip->type == 0){  // a free inode
      memset(dip, 0, sizeof(*dip));
      dip->type = type;
      bwrite(bp);   // mark it allocated on the disk
      brelse(bp);
      return iget(dev, inum);
    }
    brelse(bp);
  }
  panic("ialloc: no inodes");
}

// Copy inode, which has changed, from memory to disk.
void
iupdate(struct inode *ip)
{
  struct buf *bp;
  struct dinode *dip;

  bp = bread(ip->dev, IBLOCK(ip->inum));
  dip = (struct dinode*)bp->data + ip->inum%IPB;
  dip->type = ip->type;
  dip->major = ip->major;
  dip->minor = ip->minor;
  dip->nlink = ip->nlink;
  dip->size = ip->size;
  memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
  bwrite(bp);
  brelse(bp);
}
```

```
// Find the inode with number inum on device dev
// and return the in-memory copy.
static struct inode*
iget(uint dev, uint inum)
{
  struct inode *ip, *empty;

  acquire(&icache.lock);

  // Try for cached inode.
  empty = 0;
  for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
    if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
      ip->ref++;
      release(&icache.lock);
      return ip;
    }
    if(empty == 0 && ip->ref == 0)    // Remember empty slot.
      empty = ip;
  }

  // Allocate fresh inode.
  if(empty == 0)
    panic("iget: no inodes");

  ip = empty;
  ip->dev = dev;
  ip->inum = inum;
  ip->ref = 1;
  ip->flags = 0;
  release(&icache.lock);

  return ip;
}

// Increment reference count for ip.
// Returns ip to enable ip = idup(ip1) idiom.
struct inode*
idup(struct inode *ip)
{
  acquire(&icache.lock);
  ip->ref++;
  release(&icache.lock);
  return ip;
}
```

```
// Lock the given inode.
void
ilock(struct inode *ip)
{
  struct buf *bp;
  struct dinode *dip;

  if(ip == 0 || ip->ref < 1)
    panic("ilock");

  acquire(&icache.lock);
  while(ip->flags & I_BUSY)
    sleep(ip, &icache.lock);
  ip->flags |= I_BUSY;
  release(&icache.lock);

  if(!(ip->flags & I_VALID)){
    bp = bread(ip->dev, IBLOCK(ip->inum));
    dip = (struct dinode*)bp->data + ip->inum%IPB;
    ip->type = dip->type;
    ip->major = dip->major;
    ip->minor = dip->minor;
    ip->nlink = dip->nlink;
    ip->size = dip->size;
    memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
    brelse(bp);
    ip->flags |= I_VALID;
    if(ip->type == 0)
      panic("ilock: no type");
  }
}

// Unlock the given inode.
void
iunlock(struct inode *ip)
{
  if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
    panic("iunlock");

  acquire(&icache.lock);
  ip->flags &= ~I_BUSY;
  wakeup(ip);
  release(&icache.lock);
}
```

```
// Caller holds reference to unlocked ip.  Drop reference.
void
iput(struct inode *ip)
{
  acquire(&icache.lock);
  if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
    // inode is no longer used: truncate and free inode.
    if(ip->flags & I_BUSY)
      panic("iput busy");
    ip->flags |= I_BUSY;
    release(&icache.lock);
    itrunc(ip);
    ip->type = 0;
    iupdate(ip);
    acquire(&icache.lock);
    ip->flags = 0;
    wakeup(ip);
  }
  ip->ref--;
  release(&icache.lock);
}

// Common idiom: unlock, then put.
void
iunlockput(struct inode *ip)
{
  iunlock(ip);
  iput(ip);
}
```

```
// Inode contents
//
// The contents (data) associated with each inode is stored
// in a sequence of blocks on the disk.  The first NDIRECT blocks
// are listed in ip->addrs[].  The next NINDIRECT blocks are
// listed in the block ip->addrs[INDIRECT].

// Return the disk block address of the nth block in inode ip.
// If there is no such block, bmap allocates one.
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp;

  if(bn < NDIRECT){
    if((addr = ip->addrs[bn]) == 0)
      ip->addrs[bn] = addr = balloc(ip->dev);
    return addr;
  }
  bn -= NDIRECT;

  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = ip->addrs[NDIRECT]) == 0)
      ip->addrs[NDIRECT] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn]) == 0){
      a[bn] = addr = balloc(ip->dev);
      bwrite(bp);
    }
    brelse(bp);
    return addr;
  }

  panic("bmap: out of range");
}
```

```
// Truncate inode (discard contents).
// Only called after the last dirent referring
// to this inode has been erased on disk.
static void
itrunc(struct inode *ip)
{
  int i, j;
  struct buf *bp;
  uint *a;

  for(i = 0; i < NDIRECT; i++){
    if(ip->addrs[i]){
      bfree(ip->dev, ip->addrs[i]);
      ip->addrs[i] = 0;
    }
  }

  if(ip->addrs[NDIRECT]){
    bp = bread(ip->dev, ip->addrs[NDIRECT]);
    a = (uint*)bp->data;
    for(j = 0; j < NINDIRECT; j++){
      if(a[j])
        bfree(ip->dev, a[j]);
    }
    brelse(bp);
    bfree(ip->dev, ip->addrs[NDIRECT]);
    ip->addrs[NDIRECT] = 0;
  }

  ip->size = 0;
  iupdate(ip);
}

// Copy stat information from inode.
void
stati(struct inode *ip, struct stat *st)
{
  st->dev = ip->dev;
  st->ino = ip->inum;
  st->type = ip->type;
  st->nlink = ip->nlink;
  st->size = ip->size;
}
```

```
// Read data from inode.
int
readi(struct inode *ip, char *dst, uint off, uint n)
{
  uint tot, m;
  struct buf *bp;

  if(ip->type == T_DEV){
    if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
      return -1;
    return devsw[ip->major].read(ip, dst, n);
  }

  if(off > ip->size || off + n < off)
    return -1;
  if(off + n > ip->size)
    n = ip->size - off;

  for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(dst, bp->data + off%BSIZE, m);
    brelse(bp);
  }
  return n;
}
```

```
// Write data to inode.
int
writei(struct inode *ip, char *src, uint off, uint n)
{
  uint tot, m;
  struct buf *bp;

  if(ip->type == T_DEV){
    if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
      return -1;
    return devsw[ip->major].write(ip, src, n);
  }

  if(off > ip->size || off + n < off)
    return -1;
  if(off + n > MAXFILE*BSIZE)
    n = MAXFILE*BSIZE - off;

  for(tot=0; tot<n; tot+=m, off+=m, src+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(bp->data + off%BSIZE, src, m);
    bwrite(bp);
    brelse(bp);
  }

  if(n > 0 && off > ip->size){
    ip->size = off;
    iupdate(ip);
  }
  return n;
}
```

```
// Directories

int
namecmp(const char *s, const char *t)
{
  return strncmp(s, t, DIRSIZ);
}

// Look for a directory entry in a directory.
// If found, set *poff to byte offset of entry.
// Caller must have already locked dp.
struct inode*
dirlookup(struct inode *dp, char *name, uint *poff)
{
  uint off, inum;
  struct buf *bp;
  struct dirent *de;

  if(dp->type != T_DIR)
    panic("dirlookup not DIR");

  for(off = 0; off < dp->size; off += BSIZE){
    bp = bread(dp->dev, bmap(dp, off / BSIZE));
    for(de = (struct dirent*)bp->data;
        de < (struct dirent*)(bp->data + BSIZE);
        de++){
      if(de->inum == 0)
        continue;
      if(namecmp(name, de->name) == 0){
        // entry matches path element
        if(poff)
          *poff = off + (uchar*)de - bp->data;
        inum = de->inum;
        brelse(bp);
        return iget(dp->dev, inum);
      }
    }
    brelse(bp);
  }
  return 0;
}
```

```
// Write a new directory entry (name, inum) into the directory dp.
int
dirlink(struct inode *dp, char *name, uint inum)
{
  int off;
  struct dirent de;
  struct inode *ip;

  // Check that name is not present.
  if((ip = dirlookup(dp, name, 0)) != 0){
    iput(ip);
    return -1;
  }

  // Look for an empty dirent.
  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlink read");
    if(de.inum == 0)
      break;
  }

  strncpy(de.name, name, DIRSIZ);
  de.inum = inum;
  if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
    panic("dirlink");

  return 0;
}
```

```
// Paths

// Copy the next path element from path into name.
// Return a pointer to the element following the copied one.
// The returned path has no leading slashes,
// so the caller can check *path==’\0’ to see if the name is the last one.
// If no name to remove, return 0.
//
// Examples:
//   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
//   skipelem("///a//bb", name) = "bb", setting name = "a"
//   skipelem("a", name) = "", setting name = "a"
//   skipelem("", name) = skipelem("////", name) = 0
//
static char*
skipelem(char *path, char *name)
{
  char *s;
  int len;

  while(*path == ’/’)
    path++;
  if(*path == 0)
    return 0;
  s = path;
  while(*path != ’/’ && *path != 0)
    path++;
  len = path - s;
  if(len >= DIRSIZ)
    memmove(name, s, DIRSIZ);
  else {
    memmove(name, s, len);
    name[len] = 0;
  }
  while(*path == ’/’)
    path++;
  return path;
}
```

```
// Look up and return the inode for a path name.
// If parent != 0, return the inode for the parent and copy the final
// path element into name, which must have room for DIRSIZ bytes.
static struct inode*
namex(char *path, int nameiparent, char *name)
{
  struct inode *ip, *next;

  if(*path == ’/’)
    ip = iget(ROOTDEV, ROOTINO);
  else
    ip = idup(proc->cwd);

  while((path = skipelem(path, name)) != 0){
    ilock(ip);
    if(ip->type != T_DIR){
      iunlockput(ip);
      return 0;
    }
    if(nameiparent && *path == ’\0’){
      // Stop one level early.
      iunlock(ip);
      return ip;
    }
    if((next = dirlookup(ip, name, 0)) == 0){
      iunlockput(ip);
      return 0;
    }
    iunlockput(ip);
    ip = next;
  }
  if(nameiparent){
    iput(ip);
    return 0;
  }
  return ip;
}

struct inode*
namei(char *path)
{
  char name[DIRSIZ];
  return namex(path, 0, name);
}

struct inode*
nameiparent(char *path, char *name)
{
  return namex(path, 1, name);
}
```

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "fs.h"
#include "file.h"
#include "spinlock.h"

struct devsw devsw[NDEV];
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable;

void
fileinit(void)
{
  initlock(&ftable.lock, "ftable");
}

// Allocate a file structure.
struct file*
filealloc(void)
{
  struct file *f;

  acquire(&ftable.lock);
  for(f = ftable.file; f < ftable.file + NFILE; f++){
    if(f->ref == 0){
      f->ref = 1;
      release(&ftable.lock);
      return f;
    }
  }
  release(&ftable.lock);
  return 0;
}

// Increment ref count for file f.
struct file*
filedup(struct file *f)
{
  acquire(&ftable.lock);
  if(f->ref < 1)
    panic("filedup");
  f->ref++;
  release(&ftable.lock);
  return f;
}
```

```
// Close file f.  (Decrement ref count, close when reaches 0.)
void
fileclose(struct file *f)
{
  struct file ff;

  acquire(&ftable.lock);
  if(f->ref < 1)
    panic("fileclose");
  if(--f->ref > 0){
    release(&ftable.lock);
    return;
  }
  ff = *f;
  f->ref = 0;
  f->type = FD_NONE;
  release(&ftable.lock);

  if(ff.type == FD_PIPE)
    pipeclose(ff.pipe, ff.writable);
  else if(ff.type == FD_INODE)
    iput(ff.ip);
}

// Get metadata about file f.
int
filestat(struct file *f, struct stat *st)
{
  if(f->type == FD_INODE){
    ilock(f->ip);
    stati(f->ip, st);
    iunlock(f->ip);
    return 0;
  }
  return -1;
}
```

```
// Read from file f.  Addr is kernel address.
int
fileread(struct file *f, char *addr, int n)
{
  int r;

  if(f->readable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return piperead(f->pipe, addr, n);
  if(f->type == FD_INODE){
    ilock(f->ip);
    if((r = readi(f->ip, addr, f->off, n)) > 0)
      f->off += r;
    iunlock(f->ip);
    return r;
  }
  panic("fileread");
}

// Write to file f.  Addr is kernel address.
int
filewrite(struct file *f, char *addr, int n)
{
  int r;

  if(f->writable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return pipewrite(f->pipe, addr, n);
  if(f->type == FD_INODE){
    ilock(f->ip);
    if((r = writei(f->ip, addr, f->off, n)) > 0)
      f->off += r;
    iunlock(f->ip);
    return r;
  }
  panic("filewrite");
}
```

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "stat.h"
#include "mmu.h"
#include "proc.h"
#include "fs.h"
#include "file.h"
#include "fcntl.h"

// Fetch the nth word-sized system call argument as a file descriptor
// and return both the descriptor and the corresponding struct file.
static int
argfd(int n, int *pfd, struct file **pf)
{
  int fd;
  struct file *f;

  if(argint(n, &fd) < 0)
    return -1;
  if(fd < 0 || fd >= NOFILE || (f=proc->ofile[fd]) == 0)
    return -1;
  if(pfd)
    *pfd = fd;
  if(pf)
    *pf = f;
  return 0;
}

// Allocate a file descriptor for the given file.
// Takes over file reference from caller on success.
static int
fdalloc(struct file *f)
{
  int fd;

  for(fd = 0; fd < NOFILE; fd++){
    if(proc->ofile[fd] == 0){
      proc->ofile[fd] = f;
      return fd;
    }
  }
  return -1;
}
```

```
int
sys_dup(void)
{
  struct file *f;
  int fd;

  if(argfd(0, 0, &f) < 0)
    return -1;
  if((fd=fdalloc(f)) < 0)
    return -1;
  filedup(f);
  return fd;
}

int
sys_read(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return fileread(f, p, n);
}

int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}

int
sys_close(void)
{
  int fd;
  struct file *f;

  if(argfd(0, &fd, &f) < 0)
    return -1;
  proc->ofile[fd] = 0;
  fileclose(f);
  return 0;
}
```

```
int
sys_fstat(void)
{
  struct file *f;
  struct stat *st;

  if(argfd(0, 0, &f) < 0 || argptr(1, (void*)&st, sizeof(*st)) < 0)
    return -1;
  return filestat(f, st);
}

// Create the path new as a link to the same inode as old.
int
sys_link(void)
{
  char name[DIRSIZ], *new, *old;
  struct inode *dp, *ip;

  if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
    return -1;
  if((ip = namei(old)) == 0)
    return -1;
  ilock(ip);
  if(ip->type == T_DIR){
    iunlockput(ip);
    return -1;
  }
  ip->nlink++;
  iupdate(ip);
  iunlock(ip);

  if((dp = nameiparent(new, name)) == 0)
    goto bad;
  ilock(dp);
  if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
    iunlockput(dp);
    goto bad;
  }
  iunlockput(dp);
  iput(ip);
  return 0;

bad:
  ilock(ip);
  ip->nlink--;
  iupdate(ip);
  iunlockput(ip);
  return -1;
}
```

```
// Is the directory dp empty except for "." and ".." ?
static int
isdirempty(struct inode *dp)
{
  int off;
  struct dirent de;

  for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("isdirempty: readi");
    if(de.inum != 0)
      return 0;
  }
  return 1;
}
```

```
int
sys_unlink(void)
{
  struct inode *ip, *dp;
  struct dirent de;
  char name[DIRSIZ], *path;
  uint off;

  if(argstr(0, &path) < 0)
    return -1;
  if((dp = nameiparent(path, name)) == 0)
    return -1;
  ilock(dp);

  // Cannot unlink "." or "..".
  if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0){
    iunlockput(dp);
    return -1;
  }

  if((ip = dirlookup(dp, name, &off)) == 0){
    iunlockput(dp);
    return -1;
  }
  ilock(ip);

  if(ip->nlink < 1)
    panic("unlink: nlink < 1");
  if(ip->type == T_DIR && !isdirempty(ip)){
    iunlockput(ip);
    iunlockput(dp);
    return -1;
  }

  memset(&de, 0, sizeof(de));
  if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
    panic("unlink: writei");
  if(ip->type == T_DIR){
    dp->nlink--;
    iupdate(dp);
  }
  iunlockput(dp);

  ip->nlink--;
  iupdate(ip);
  iunlockput(ip);
  return 0;
}
```

```c
static struct inode*
create(char *path, short type, short major, short minor)
{
  uint off;
  struct inode *ip, *dp;
  char name[DIRSIZ];

  if((dp = nameiparent(path, name)) == 0)
    return 0;
  ilock(dp);

  if((ip = dirlookup(dp, name, &off)) != 0){
    iunlockput(dp);
    ilock(ip);
    if(type == T_FILE && ip->type == T_FILE)
      return ip;
    iunlockput(ip);
    return 0;
  }

  if((ip = ialloc(dp->dev, type)) == 0)
    panic("create: ialloc");

  ilock(ip);
  ip->major = major;
  ip->minor = minor;
  ip->nlink = 1;
  iupdate(ip);

  if(type == T_DIR){  // Create . and .. entries.
    dp->nlink++;  // for ".."
    iupdate(dp);
    // No ip->nlink++ for ".": avoid cyclic ref count.
    if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
      panic("create dots");
  }

  if(dirlink(dp, name, ip->inum) < 0)
    panic("create: dirlink");

  iunlockput(dp);
  return ip;
}
```

```c
int
sys_open(void)
{
  char *path;
  int fd, omode;
  struct file *f;
  struct inode *ip;

  if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
    return -1;

  if(omode & O_CREATE){
    if((ip = create(path, T_FILE, 0, 0)) == 0)
      return -1;
  } else {
    if((ip = namei(path)) == 0)
      return -1;
    ilock(ip);
    if(ip->type == T_DIR && omode != O_RDONLY){
      iunlockput(ip);
      return -1;
    }
  }

  if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
    if(f)
      fileclose(f);
    iunlockput(ip);
    return -1;
  }
  iunlock(ip);

  f->type = FD_INODE;
  f->ip = ip;
  f->off = 0;
  f->readable = !(omode & O_WRONLY);
  f->writable = (omode & O_WRONLY) || (omode & O_RDWR);

  return fd;
}
```

```
int
sys_mkdir(void)
{
  char *path;
  struct inode *ip;

  if(argstr(0, &path) < 0 || (ip = create(path, T_DIR, 0, 0)) == 0)
    return -1;
  iunlockput(ip);
  return 0;
}

int
sys_mknod(void)
{
  struct inode *ip;
  char *path;
  int len;
  int major, minor;

  if((len=argstr(0, &path)) < 0 ||
     argint(1, &major) < 0 ||
     argint(2, &minor) < 0 ||
     (ip = create(path, T_DEV, major, minor)) == 0)
    return -1;
  iunlockput(ip);
  return 0;
}

int
sys_chdir(void)
{
  char *path;
  struct inode *ip;

  if(argstr(0, &path) < 0 || (ip = namei(path)) == 0)
    return -1;
  ilock(ip);
  if(ip->type != T_DIR){
    iunlockput(ip);
    return -1;
  }
  iunlock(ip);
  iput(proc->cwd);
  proc->cwd = ip;
  return 0;
}
```

```
int
sys_exec(void)
{
  char *path, *argv[20];
  int i;
  uint uargv, uarg;

  if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0)
    return -1;
  memset(argv, 0, sizeof(argv));
  for(i=0;; i++){
    if(i >= NELEM(argv))
      return -1;
    if(fetchint(proc, uargv+4*i, (int*)&uarg) < 0)
      return -1;
    if(uarg == 0){
      argv[i] = 0;
      break;
    }
    if(fetchstr(proc, uarg, &argv[i]) < 0)
      return -1;
  }
  return exec(path, argv);
}

int
sys_pipe(void)
{
  int *fd;
  struct file *rf, *wf;
  int fd0, fd1;

  if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
    return -1;
  if(pipealloc(&rf, &wf) < 0)
    return -1;
  fd0 = -1;
  if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
    if(fd0 >= 0)
      proc->ofile[fd0] = 0;
    fileclose(rf);
    fileclose(wf);
    return -1;
  }
  fd[0] = fd0;
  fd[1] = fd1;
  return 0;
}
```

```
#include "types.h"
#include "param.h"
#include "mmu.h"
#include "proc.h"
#include "defs.h"
#include "x86.h"
#include "elf.h"

int
exec(char *path, char **argv)
{
  char *mem, *s, *last;
  int i, argc, arglen, len, off;
  uint sz, sp, argp;
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;

  mem = 0;
  sz = 0;

  if((ip = namei(path)) == 0)
    return -1;
  ilock(ip);

  // Check ELF header
  if(readi(ip, (char*)&elf, 0, sizeof(elf)) < sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;

  // Compute memory size of new process.
  // Program segments.
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)
      goto bad;
    sz += ph.memsz;
  }

  // Arguments.
  arglen = 0;
  for(argc=0; argv[argc]; argc++)
    arglen += strlen(argv[argc]) + 1;
  arglen = (arglen+3) & ~3;
  sz += arglen;
  sz += 4*(argc+1);  // argv data
```

```
  sz += 4;  // argv
  sz += 4;  // argc

  // Stack.
  sz += PAGE;

  // Allocate program memory.
  sz = (sz+PAGE-1) & ~(PAGE-1);
  mem = kalloc(sz);
  if(mem == 0)
    goto bad;
  memset(mem, 0, sz);

  // Load program into memory.
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.va + ph.memsz < ph.va || ph.va + ph.memsz > sz)
      goto bad;
    if(ph.memsz < ph.filesz)
      goto bad;
    if(readi(ip, mem + ph.va, ph.offset, ph.filesz) != ph.filesz)
      goto bad;
    memset(mem + ph.va + ph.filesz, 0, ph.memsz - ph.filesz);
  }
  iunlockput(ip);

  // Initialize stack.
  sp = sz;
  argp = sz - arglen - 4*(argc+1);

  // Copy argv strings and pointers to stack.
  *(uint*)(mem+argp + 4*argc) = 0;  // argv[argc]
  for(i=argc-1; i>=0; i--){
    len = strlen(argv[i]) + 1;
    sp -= len;
    memmove(mem+sp, argv[i], len);
    *(uint*)(mem+argp + 4*i) = sp;  // argv[i]
  }

  // Stack frame for main(argc, argv), below arguments.
  sp = argp;
  sp -= 4;
  *(uint*)(mem+sp) = argp;
  sp -= 4;
  *(uint*)(mem+sp) = argc;
  sp -= 4;
  *(uint*)(mem+sp) = 0xffffffff;   // fake return pc
  // Save program name for debugging.
  for(last=s=path; *s; s++)
    if(*s == ’/’)
      last = s+1;
  safestrcpy(proc->name, last, sizeof(proc->name));

  // Commit to the new image.
  kfree(proc->mem, proc->sz);
  proc->mem = mem;
  proc->sz = sz;
  proc->tf->eip = elf.entry;  // main
  proc->tf->esp = sp;
  usegment();
  return 0;

 bad:
  if(mem)
    kfree(mem, sz);
  iunlockput(ip);
  return -1;
}
```

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "mmu.h"
#include "proc.h"
#include "fs.h"
#include "file.h"
#include "spinlock.h"

#define PIPESIZE 512

struct pipe {
  struct spinlock lock;
  char data[PIPESIZE];
  uint nread;     // number of bytes read
  uint nwrite;    // number of bytes written
  int readopen;   // read fd is still open
  int writeopen;  // write fd is still open
};

int
pipealloc(struct file **f0, struct file **f1)
{
  struct pipe *p;

  p = 0;
  *f0 = *f1 = 0;
  if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
    goto bad;
  if((p = (struct pipe*)kalloc(PAGE)) == 0)
    goto bad;
  p->readopen = 1;
  p->writeopen = 1;
  p->nwrite = 0;
  p->nread = 0;
  initlock(&p->lock, "pipe");
  (*f0)->type = FD_PIPE;
  (*f0)->readable = 1;
  (*f0)->writable = 0;
  (*f0)->pipe = p;
  (*f1)->type = FD_PIPE;
  (*f1)->readable = 0;
  (*f1)->writable = 1;
  (*f1)->pipe = p;
  return 0;
```

```
 bad:
  if(p)
    kfree((char*)p, PAGE);
  if(*f0)
    fileclose(*f0);
  if(*f1)
    fileclose(*f1);
  return -1;
}

void
pipeclose(struct pipe *p, int writable)
{
  acquire(&p->lock);
  if(writable){
    p->writeopen = 0;
    wakeup(&p->nread);
  } else {
    p->readopen = 0;
    wakeup(&p->nwrite);
  }
  if(p->readopen == 0 && p->writeopen == 0) {
    release(&p->lock);
    kfree((char*)p, PAGE);
  } else
    release(&p->lock);
}


int
pipewrite(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  for(i = 0; i < n; i++){
    while(p->nwrite == p->nread + PIPESIZE) {
      if(p->readopen == 0 || proc->killed){
        release(&p->lock);
        return -1;
      }
      wakeup(&p->nread);
      sleep(&p->nwrite, &p->lock);
    }
    p->data[p->nwrite++ % PIPESIZE] = addr[i];
  }
  wakeup(&p->nread);
  release(&p->lock);
  return n;
}
```

```
int
piperead(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  while(p->nread == p->nwrite && p->writeopen){
    if(proc->killed){
      release(&p->lock);
      return -1;
    }
    sleep(&p->nread, &p->lock);
  }
  for(i = 0; i < n; i++){
    if(p->nread == p->nwrite)
      break;
    addr[i] = p->data[p->nread++ % PIPESIZE];
  }
  wakeup(&p->nwrite);
  release(&p->lock);
  return i;
}
```

```c
#include "types.h"
#include "x86.h"

void*
memset(void *dst, int c, uint n)
{
  stosb(dst, c, n);
  return dst;
}

int
memcmp(const void *v1, const void *v2, uint n)
{
  const uchar *s1, *s2;

  s1 = v1;
  s2 = v2;
  while(n-- > 0){
    if(*s1 != *s2)
      return *s1 - *s2;
    s1++, s2++;
  }

  return 0;
}

void*
memmove(void *dst, const void *src, uint n)
{
  const char *s;
  char *d;

  s = src;
  d = dst;
  if(s < d && s + n > d){
    s += n;
    d += n;
    while(n-- > 0)
      *--d = *--s;
  } else
    while(n-- > 0)
      *d++ = *s++;

  return dst;
}
```

```c
int
strncmp(const char *p, const char *q, uint n)
{
  while(n > 0 && *p && *p == *q)
    n--, p++, q++;
  if(n == 0)
    return 0;
  return (uchar)*p - (uchar)*q;
}

char*
strncpy(char *s, const char *t, int n)
{
  char *os;

  os = s;
  while(n-- > 0 && (*s++ = *t++) != 0)
    ;
  while(n-- > 0)
    *s++ = 0;
  return os;
}

// Like strncpy but guaranteed to NUL-terminate.
char*
safestrcpy(char *s, const char *t, int n)
{
  char *os;

  os = s;
  if(n <= 0)
    return os;
  while(--n > 0 && (*s++ = *t++) != 0)
    ;
  *s = 0;
  return os;
}

int
strlen(const char *s)
{
  int n;

  for(n = 0; s[n]; n++)
    ;
  return n;
}
```

```
// See MultiProcessor Specification Version 1.[14]

struct mp {             // floating pointer
  uchar signature[4];           // "_MP_"
  void *physaddr;               // phys addr of MP config table
  uchar length;                 // 1
  uchar specrev;                // [14]
  uchar checksum;               // all bytes must add up to 0
  uchar type;                   // MP system config type
  uchar imcrp;
  uchar reserved[3];
};

struct mpconf {         // configuration table header
  uchar signature[4];           // "PCMP"
  ushort length;                // total table length
  uchar version;                // [14]
  uchar checksum;               // all bytes must add up to 0
  uchar product[20];            // product id
  uint *oemtable;               // OEM table pointer
  ushort oemlength;             // OEM table length
  ushort entry;                 // entry count
  uint *lapicaddr;              // address of local APIC
  ushort xlength;               // extended table length
  uchar xchecksum;              // extended table checksum
  uchar reserved;
};

struct mpproc {         // processor table entry
  uchar type;                   // entry type (0)
  uchar apicid;                 // local APIC id
  uchar version;                // local APIC verison
  uchar flags;                  // CPU flags
    #define MPBOOT 0x02           // This proc is the bootstrap processor.
  uchar signature[4];           // CPU signature
  uint feature;                 // feature flags from CPUID instruction
  uchar reserved[8];
};

struct mpioapic {       // I/O APIC table entry
  uchar type;                   // entry type (2)
  uchar apicno;                 // I/O APIC id
  uchar version;                // I/O APIC version
  uchar flags;                  // I/O APIC flags
  uint *addr;                  // I/O APIC address
};
```

```
// Table entry types
#define MPPROC    0x00  // One per processor
#define MPBUS     0x01  // One per bus
#define MPIOAPIC  0x02  // One per I/O APIC
#define MPIOINTR  0x03  // One per bus interrupt source
#define MPLINTR   0x04  // One per system interrupt source
```

```c
// Multiprocessor bootstrap.
// Search memory for MP description structures.
// http://developer.intel.com/design/pentium/datashts/24201606.pdf

#include "types.h"
#include "defs.h"
#include "param.h"
#include "mp.h"
#include "x86.h"
#include "mmu.h"
#include "proc.h"

struct cpu cpus[NCPU];
static struct cpu *bcpu;
int ismp;
int ncpu;
uchar ioapicid;

int
mpbcpu(void)
{
  return bcpu-cpus;
}

static uchar
sum(uchar *addr, int len)
{
  int i, sum;

  sum = 0;
  for(i=0; i<len; i++)
    sum += addr[i];
  return sum;
}

// Look for an MP structure in the len bytes at addr.
static struct mp*
mpsearch1(uchar *addr, int len)
{
  uchar *e, *p;

  e = addr+len;
  for(p = addr; p < e; p += sizeof(struct mp))
    if(memcmp(p, "_MP_", 4) == 0 && sum(p, sizeof(struct mp)) == 0)
      return (struct mp*)p;
  return 0;
}
```

```c
// Search for the MP Floating Pointer Structure, which according to the
// spec is in one of the following three locations:
// 1) in the first KB of the EBDA;
// 2) in the last KB of system base memory;
// 3) in the BIOS ROM between 0xE0000 and 0xFFFFF.
static struct mp*
mpsearch(void)
{
  uchar *bda;
  uint p;
  struct mp *mp;

  bda = (uchar*)0x400;
  if((p = ((bda[0x0F]<<8)|bda[0x0E]) << 4)){
    if((mp = mpsearch1((uchar*)p, 1024)))
      return mp;
  } else {
    p = ((bda[0x14]<<8)|bda[0x13])*1024;
    if((mp = mpsearch1((uchar*)p-1024, 1024)))
      return mp;
  }
  return mpsearch1((uchar*)0xF0000, 0x10000);
}

// Search for an MP configuration table.  For now,
// don't accept the default configurations (physaddr == 0).
// Check for correct signature, calculate the checksum and,
// if correct, check the version.
// To do: check extended table checksum.
static struct mpconf*
mpconfig(struct mp **pmp)
{
  struct mpconf *conf;
  struct mp *mp;

  if((mp = mpsearch()) == 0 || mp->physaddr == 0)
    return 0;
  conf = (struct mpconf*)mp->physaddr;
  if(memcmp(conf, "PCMP", 4) != 0)
    return 0;
  if(conf->version != 1 && conf->version != 4)
    return 0;
  if(sum((uchar*)conf, conf->length) != 0)
    return 0;
  *pmp = mp;
  return conf;
}
```

```
void
mpinit(void)
{
  uchar *p, *e;
  struct mp *mp;
  struct mpconf *conf;
  struct mpproc *proc;
  struct mpioapic *ioapic;

  bcpu = &cpus[0];
  if((conf = mpconfig(&mp)) == 0)
    return;
  ismp = 1;
  lapic = (uint*)conf->lapicaddr;
  for(p=(uchar*)(conf+1), e=(uchar*)conf+conf->length; p<e; ){
    switch(*p){
    case MPPROC:
      proc = (struct mpproc*)p;
      if(ncpu != proc->apicid) {
        cprintf("mpinit: ncpu=%d apicpid=%d", ncpu, proc->apicid);
        panic("mpinit");
      }
      if(proc->flags & MPBOOT)
        bcpu = &cpus[ncpu];
      cpus[ncpu].id = ncpu;
      ncpu++;
      p += sizeof(struct mpproc);
      continue;
    case MPIOAPIC:
      ioapic = (struct mpioapic*)p;
      ioapicid = ioapic->apicno;
      p += sizeof(struct mpioapic);
      continue;
    case MPBUS:
    case MPIOINTR:
    case MPLINTR:
      p += 8;
      continue;
    default:
      cprintf("mpinit: unknown config type %x\n", *p);
      panic("mpinit");
    }
  }
  if(mp->imcrp){
    // Bochs doesn't support IMCR, so this doesn't run on Bochs.
    // But it would on real hardware.
    outb(0x22, 0x70);   // Select IMCR
    outb(0x23, inb(0x23) | 1);  // Mask external interrupts.
  }
}
```

```
// The local APIC manages internal (non-I/O) interrupts.
// See Chapter 8 & Appendix C of Intel processor manual volume 3.

#include "types.h"
#include "defs.h"
#include "traps.h"
#include "mmu.h"
#include "x86.h"

// Local APIC registers, divided by 4 for use as uint[] indices.
#define ID      (0x0020/4)   // ID
#define VER     (0x0030/4)   // Version
#define TPR     (0x0080/4)   // Task Priority
#define EOI     (0x00B0/4)   // EOI
#define SVR     (0x00F0/4)   // Spurious Interrupt Vector
  #define ENABLE     0x00000100   // Unit Enable
#define ESR     (0x0280/4)   // Error Status
#define ICRLO   (0x0300/4)   // Interrupt Command
  #define INIT       0x00000500   // INIT/RESET
  #define STARTUP    0x00000600   // Startup IPI
  #define DELIVS     0x00001000   // Delivery status
  #define ASSERT     0x00004000   // Assert interrupt (vs deassert)
  #define LEVEL      0x00008000   // Level triggered
  #define BCAST      0x00080000   // Send to all APICs, including self.
#define ICRHI   (0x0310/4)   // Interrupt Command [63:32]
#define TIMER   (0x0320/4)   // Local Vector Table 0 (TIMER)
  #define X1         0x0000000B   // divide counts by 1
  #define PERIODIC   0x00020000   // Periodic
#define PCINT   (0x0340/4)   // Performance Counter LVT
#define LINT0   (0x0350/4)   // Local Vector Table 1 (LINT0)
#define LINT1   (0x0360/4)   // Local Vector Table 2 (LINT1)
#define ERROR   (0x0370/4)   // Local Vector Table 3 (ERROR)
  #define MASKED     0x00010000   // Interrupt masked
#define TICR    (0x0380/4)   // Timer Initial Count
#define TCCR    (0x0390/4)   // Timer Current Count
#define TDCR    (0x03E0/4)   // Timer Divide Configuration

volatile uint *lapic;  // Initialized in mp.c

static void
lapicw(int index, int value)
{
  lapic[index] = value;
  lapic[ID];  // wait for write to finish, by reading
}
```

```
void
lapicinit(int c)
{
  if(!lapic)
    return;

  // Enable local APIC; set spurious interrupt vector.
  lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));

  // The timer repeatedly counts down at bus frequency
  // from lapic[TICR] and then issues an interrupt.
  // If xv6 cared more about precise timekeeping,
  // TICR would be calibrated using an external time source.
  lapicw(TDCR, X1);
  lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
  lapicw(TICR, 10000000);

  // Disable logical interrupt lines.
  lapicw(LINT0, MASKED);
  lapicw(LINT1, MASKED);

  // Disable performance counter overflow interrupts
  // on machines that provide that interrupt entry.
  if(((lapic[VER]>>16) & 0xFF) >= 4)
    lapicw(PCINT, MASKED);

  // Map error interrupt to IRQ_ERROR.
  lapicw(ERROR, T_IRQ0 + IRQ_ERROR);

  // Clear error status register (requires back-to-back writes).
  lapicw(ESR, 0);
  lapicw(ESR, 0);

  // Ack any outstanding interrupts.
  lapicw(EOI, 0);

  // Send an Init Level De-Assert to synchronise arbitration ID’s.
  lapicw(ICRHI, 0);
  lapicw(ICRLO, BCAST | INIT | LEVEL);
  while(lapic[ICRLO] & DELIVS)
    ;

  // Enable interrupts on the APIC (but not on the processor).
  lapicw(TPR, 0);
}
```

```
int
cpunum(void)
{
  // Cannot call cpu when interrupts are enabled:
  // result not guaranteed to last long enough to be used!
  // Would prefer to panic but even printing is chancy here:
  // almost everything, including cprintf and panic, calls cpu,
  // often indirectly through acquire and release.
  if(readeflags()&FL_IF){
    static int n;
    if(n++ == 0)
      cprintf("cpu called from %x with interrupts enabled\n",
        __builtin_return_address(0));
  }

  if(lapic)
    return lapic[ID]>>24;
  return 0;
}

// Acknowledge interrupt.
void
lapiceoi(void)
{
  if(lapic)
    lapicw(EOI, 0);
}

// Spin for a given number of microseconds.
// On real hardware would want to tune this dynamically.
void
microdelay(int us)
{
}


#define IO_RTC  0x70

// Start additional processor running bootstrap code at addr.
// See Appendix B of MultiProcessor Specification.
void
lapicstartap(uchar apicid, uint addr)
{
  int i;
  ushort *wrv;

  // "The BSP must initialize CMOS shutdown code to 0AH
  // and the warm reset vector (DWORD based at 40:67) to point at
  // the AP startup code prior to the [universal startup algorithm]."
  outb(IO_RTC, 0xF);  // offset 0xF is shutdown code
  outb(IO_RTC+1, 0x0A);
  wrv = (ushort*)(0x40<<4 | 0x67);  // Warm reset vector
  wrv[0] = 0;
  wrv[1] = addr >> 4;

  // "Universal startup algorithm."
  // Send INIT (level-triggered) interrupt to reset other CPU.
  lapicw(ICRHI, apicid<<24);
  lapicw(ICRLO, INIT | LEVEL | ASSERT);
  microdelay(200);
  lapicw(ICRLO, INIT | LEVEL);
  microdelay(100);    // should be 10ms, but too slow in Bochs!

  // Send startup IPI (twice!) to enter bootstrap code.
  // Regular hardware is supposed to only accept a STARTUP
  // when it is in the halted state due to an INIT.  So the second
  // should be ignored, but it is part of the official Intel algorithm.
  // Bochs complains about the second one.  Too bad for Bochs.
  for(i = 0; i < 2; i++){
    lapicw(ICRHI, apicid<<24);
    lapicw(ICRLO, STARTUP | (addr>>12));
    microdelay(200);
  }
}
```

```
// The I/O APIC manages hardware interrupts for an SMP system.
// http://www.intel.com/design/chipsets/datashts/29056601.pdf
// See also picirq.c.

#include "types.h"
#include "defs.h"
#include "traps.h"

#define IOAPIC  0xFEC00000   // Default physical address of IO APIC

#define REG_ID     0x00  // Register index: ID
#define REG_VER    0x01  // Register index: version
#define REG_TABLE  0x10  // Redirection table base

// The redirection table starts at REG_TABLE and uses
// two registers to configure each interrupt.
// The first (low) register in a pair contains configuration bits.
// The second (high) register contains a bitmask telling which
// CPUs can serve that interrupt.
#define INT_DISABLED   0x00010000  // Interrupt disabled
#define INT_LEVEL      0x00008000  // Level-triggered (vs edge-)
#define INT_ACTIVELOW  0x00002000  // Active low (vs high)
#define INT_LOGICAL    0x00000800  // Destination is CPU id (vs APIC ID)

volatile struct ioapic *ioapic;

// IO APIC MMIO structure: write reg, then read or write data.
struct ioapic {
  uint reg;
  uint pad[3];
  uint data;
};

static uint
ioapicread(int reg)
{
  ioapic->reg = reg;
  return ioapic->data;
}

static void
ioapicwrite(int reg, uint data)
{
  ioapic->reg = reg;
  ioapic->data = data;
}
```

```c
void
ioapicinit(void)
{
  int i, id, maxintr;

  if(!ismp)
    return;

  ioapic = (volatile struct ioapic*)IOAPIC;
  maxintr = (ioapicread(REG_VER) >> 16) & 0xFF;
  id = ioapicread(REG_ID) >> 24;
  if(id != ioapicid)
    cprintf("ioapicinit: id isn’t equal to ioapicid; not a MP\n");

  // Mark all interrupts edge-triggered, active high, disabled,
  // and not routed to any CPUs.
  for(i = 0; i <= maxintr; i++){
    ioapicwrite(REG_TABLE+2*i, INT_DISABLED | (T_IRQ0 + i));
    ioapicwrite(REG_TABLE+2*i+1, 0);
  }
}

void
ioapicenable(int irq, int cpunum)
{
  if(!ismp)
    return;

  // Mark interrupt edge-triggered, active high,
  // enabled, and routed to the given cpunum,
  // which happens to be that cpu’s APIC ID.
  ioapicwrite(REG_TABLE+2*irq, T_IRQ0 + irq);
  ioapicwrite(REG_TABLE+2*irq+1, cpunum << 24);
}
```

```c
// Intel 8259A programmable interrupt controllers.

#include "types.h"
#include "x86.h"
#include "traps.h"

// I/O Addresses of the two programmable interrupt controllers
#define IO_PIC1         0x20    // Master (IRQs 0-7)
#define IO_PIC2         0xA0    // Slave (IRQs 8-15)

#define IRQ_SLAVE       2       // IRQ at which slave connects to master

// Current IRQ mask.
// Initial IRQ mask has interrupt 2 enabled (for slave 8259A).
static ushort irqmask = 0xFFFF & ~(1<<IRQ_SLAVE);

static void
picsetmask(ushort mask)
{
  irqmask = mask;
  outb(IO_PIC1+1, mask);
  outb(IO_PIC2+1, mask >> 8);
}

void
picenable(int irq)
{
  picsetmask(irqmask & ~(1<<irq));
}

// Initialize the 8259A interrupt controllers.
void
picinit(void)
{
  // mask all interrupts
  outb(IO_PIC1+1, 0xFF);
  outb(IO_PIC2+1, 0xFF);

  // Set up master (8259A-1)

  // ICW1:  0001g0hi
  //    g:  0 = edge triggering, 1 = level triggering
  //    h:  0 = cascaded PICs, 1 = master only
  //    i:  0 = no ICW4, 1 = ICW4 required
  outb(IO_PIC1, 0x11);

  // ICW2:  Vector offset
  outb(IO_PIC1+1, T_IRQ0);
  // ICW3:  (master PIC) bit mask of IR lines connected to slaves
  //        (slave PIC) 3-bit # of slave's connection to master
  outb(IO_PIC1+1, 1<<IRQ_SLAVE);

  // ICW4:  000nbmap
  //    n:  1 = special fully nested mode
  //    b:  1 = buffered mode
  //    m:  0 = slave PIC, 1 = master PIC
  //      (ignored when b is 0, as the master/slave role
  //      can be hardwired).
  //    a:  1 = Automatic EOI mode
  //    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
  outb(IO_PIC1+1, 0x3);

  // Set up slave (8259A-2)
  outb(IO_PIC2, 0x11);                  // ICW1
  outb(IO_PIC2+1, T_IRQ0 + 8);      // ICW2
  outb(IO_PIC2+1, IRQ_SLAVE);           // ICW3
  // NB Automatic EOI mode doesn't tend to work on the slave.
  // Linux source code says it's "to be investigated".
  outb(IO_PIC2+1, 0x3);                 // ICW4

  // OCW3:  0ef01prs
  //   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
  //    p:  0 = no polling, 1 = polling mode
  //   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
  outb(IO_PIC1, 0x68);             // clear specific mask
  outb(IO_PIC1, 0x0a);             // read IRR by default

  outb(IO_PIC2, 0x68);             // OCW3
  outb(IO_PIC2, 0x0a);             // OCW3

  if(irqmask != 0xFFFF)
    picsetmask(irqmask);
}
```

```
// Blank page.
// PC keyboard interface constants

#define KBSTATP         0x64    // kbd controller status port(I)
#define KBS_DIB         0x01    // kbd data in buffer
#define KBDATAP         0x60    // kbd data port(I)

#define NO              0

#define SHIFT           (1<<0)
#define CTL             (1<<1)
#define ALT             (1<<2)

#define CAPSLOCK        (1<<3)
#define NUMLOCK         (1<<4)
#define SCROLLLOCK      (1<<5)

#define E0ESC           (1<<6)

// Special keycodes
#define KEY_HOME        0xE0
#define KEY_END         0xE1
#define KEY_UP          0xE2
#define KEY_DN          0xE3
#define KEY_LF          0xE4
#define KEY_RT          0xE5
#define KEY_PGUP        0xE6
#define KEY_PGDN        0xE7
#define KEY_INS         0xE8
#define KEY_DEL         0xE9

// C(’A’) == Control-A
#define C(x) (x - ’@’)

static uchar shiftcode[256] =
{
  [0x1D] CTL,
  [0x2A] SHIFT,
  [0x36] SHIFT,
  [0x38] ALT,
  [0x9D] CTL,
  [0xB8] ALT
};

static uchar togglecode[256] =
{
  [0x3A] CAPSLOCK,
  [0x45] NUMLOCK,
  [0x46] SCROLLLOCK
};

```

```
static uchar normalmap[256] =
{
  NO,   0x1B, ’1’,  ’2’,  ’3’,  ’4’,  ’5’,  ’6’,  // 0x00
  ’7’,  ’8’,  ’9’,  ’0’,  ’-’,  ’=’,  ’\b’, ’\t’,
  ’q’,  ’w’,  ’e’,  ’r’,  ’t’,  ’y’,  ’u’,  ’i’,  // 0x10
  ’o’,  ’p’,  ’[’,  ’]’,  ’\n’, NO,   ’a’,  ’s’,
  ’d’,  ’f’,  ’g’,  ’h’,  ’j’,  ’k’,  ’l’,  ’;’,  // 0x20
  ’\’’, ’‘’,  NO,   ’\\’, ’z’,  ’x’,  ’c’,  ’v’,
  ’b’,  ’n’,  ’m’,  ’,’,  ’.’,  ’/’,  NO,   ’*’,  // 0x30
  NO,   ’ ’,  NO,   NO,   NO,   NO,   NO,   NO,
  NO,   NO,   NO,   NO,   NO,   NO,   NO,   ’7’,  // 0x40
  ’8’,  ’9’,  ’-’,  ’4’,  ’5’,  ’6’,  ’+’,  ’1’,
  ’2’,  ’3’,  ’0’,  ’.’,  NO,   NO,   NO,   NO,   // 0x50
  [0x9C] ’\n’,      // KP_Enter
  [0xB5] ’/’,       // KP_Div
  [0xC8] KEY_UP,    [0xD0] KEY_DN,
  [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
  [0xCB] KEY_LF,    [0xCD] KEY_RT,
  [0x97] KEY_HOME,  [0xCF] KEY_END,
  [0xD2] KEY_INS,   [0xD3] KEY_DEL
};

static uchar shiftmap[256] =
{
  NO,   033,  ’!’,  ’@’,  ’#’,  ’$’,  ’%’,  ’^’,  // 0x00
  ’&’,  ’*’,  ’(’,  ’)’,  ’_’,  ’+’,  ’\b’, ’\t’,
  ’Q’,  ’W’,  ’E’,  ’R’,  ’T’,  ’Y’,  ’U’,  ’I’,  // 0x10
  ’O’,  ’P’,  ’{’,  ’}’,  ’\n’, NO,   ’A’,  ’S’,
  ’D’,  ’F’,  ’G’,  ’H’,  ’J’,  ’K’,  ’L’,  ’:’,  // 0x20
  ’"’,  ’~’,  NO,   ’|’,  ’Z’,  ’X’,  ’C’,  ’V’,
  ’B’,  ’N’,  ’M’,  ’<’,  ’>’,  ’?’,  NO,   ’*’,  // 0x30
  NO,   ’ ’,  NO,   NO,   NO,   NO,   NO,   NO,
  NO,   NO,   NO,   NO,   NO,   NO,   NO,   ’7’,  // 0x40
  ’8’,  ’9’,  ’-’,  ’4’,  ’5’,  ’6’,  ’+’,  ’1’,
  ’2’,  ’3’,  ’0’,  ’.’,  NO,   NO,   NO,   NO,   // 0x50
  [0x9C] ’\n’,      // KP_Enter
  [0xB5] ’/’,       // KP_Div
  [0xC8] KEY_UP,    [0xD0] KEY_DN,
  [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
  [0xCB] KEY_LF,    [0xCD] KEY_RT,
  [0x97] KEY_HOME,  [0xCF] KEY_END,
  [0xD2] KEY_INS,   [0xD3] KEY_DEL
};

```

```
static uchar ctlmap[256] =
{
  NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
  NO,      NO,      NO,      NO,      NO,      NO,      NO,      NO,
  C('Q'),  C('W'),  C('E'),  C('R'),  C('T'),  C('Y'),  C('U'),  C('I'),
  C('O'),  C('P'),  NO,      NO,      '\r',    NO,      C('A'),  C('S'),
  C('D'),  C('F'),  C('G'),  C('H'),  C('J'),  C('K'),  C('L'),  NO,
  NO,      NO,      NO,      C('\\'), C('Z'),  C('X'),  C('C'),  C('V'),
  C('B'),  C('N'),  C('M'),  NO,      NO,      C('/'),  NO,      NO,
  [0x9C] '\r',      // KP_Enter
  [0xB5] C('/'),    // KP_Div
  [0xC8] KEY_UP,    [0xD0] KEY_DN,
  [0xC9] KEY_PGUP,  [0xD1] KEY_PGDN,
  [0xCB] KEY_LF,    [0xCD] KEY_RT,
  [0x97] KEY_HOME,  [0xCF] KEY_END,
  [0xD2] KEY_INS,   [0xD3] KEY_DEL
};
```

```
#include "types.h"
#include "x86.h"
#include "defs.h"
#include "kbd.h"

int
kbdgetc(void)
{
  static uint shift;
  static uchar *charcode[4] = {
    normalmap, shiftmap, ctlmap, ctlmap
  };
  uint st, data, c;

  st = inb(KBSTATP);
  if((st & KBS_DIB) == 0)
    return -1;
  data = inb(KBDATAP);

  if(data == 0xE0){
    shift |= E0ESC;
    return 0;
  } else if(data & 0x80){
    // Key released
    data = (shift & E0ESC ? data : data & 0x7F);
    shift &= ~(shiftcode[data] | E0ESC);
    return 0;
  } else if(shift & E0ESC){
    // Last character was an E0 escape; or with 0x80
    data |= 0x80;
    shift &= ~E0ESC;
  }

  shift |= shiftcode[data];
  shift ^= togglecode[data];
  c = charcode[shift & (CTL | SHIFT)][data];
  if(shift & CAPSLOCK){
    if('a' <= c && c <= 'z')
      c += 'A' - 'a';
    else if('A' <= c && c <= 'Z')
      c += 'a' - 'A';
  }
  return c;
}

void
kbdintr(void)
{
  consoleintr(kbdgetc);
}
```

```
// Console input and output.
// Input is from the keyboard or serial port.
// Output is written to the screen and serial port.

#include "types.h"
#include "defs.h"
#include "param.h"
#include "traps.h"
#include "spinlock.h"
#include "fs.h"
#include "file.h"
#include "mmu.h"
#include "proc.h"
#include "x86.h"

static void consputc(int);

static int panicked = 0;

static struct {
   struct spinlock lock;
   int locking;
} cons;

static void
printint(int xx, int base, int sgn)
{
  static char digits[] = "0123456789abcdef";
  char buf[16];
  int i = 0, neg = 0;
  uint x;

  if(sgn && xx < 0){
    neg = 1;
    x = -xx;
  } else
    x = xx;

  do{
    buf[i++] = digits[x % base];
  }while((x /= base) != 0);
  if(neg)
    buf[i++] = ’-’;

  while(--i >= 0)
    consputc(buf[i]);
}
```

```
// Print to the console. only understands %d, %x, %p, %s.
void
cprintf(char *fmt, ...)
{
  int i, c, state, locking;
  uint *argp;
  char *s;

  locking = cons.locking;
  if(locking)
    acquire(&cons.lock);

  argp = (uint*)(void*)(&fmt + 1);
  state = 0;
  for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
    if(c != ’%’){
      consputc(c);
      continue;
    }
    c = fmt[++i] & 0xff;
    if(c == 0)
      break;
    switch(c){
    case ’d’:
      printint(*argp++, 10, 1);
      break;
    case ’x’:
    case ’p’:
      printint(*argp++, 16, 0);
      break;
    case ’s’:
      if((s = (char*)*argp++) == 0)
        s = "(null)";
      for(; *s; s++)
        consputc(*s);
      break;
    case ’%’:
      consputc(’%’);
      break;
    default:
      // Print unknown % sequence to draw attention.
      consputc(’%’);
      consputc(c);
      break;
    }
  }

  if(locking)
    release(&cons.lock);
}
```

```
void
panic(char *s)
{
  int i;
  uint pcs[10];

  cli();
  cons.locking = 0;
  cprintf("cpu%d: panic: ", cpu->id);
  cprintf(s);
  cprintf("\n");
  getcallerpcs(&s, pcs);
  for(i=0; i<10; i++)
    cprintf(" %p", pcs[i]);
  panicked = 1; // freeze other CPU
  for(;;)
    ;
}
```

```
#define BACKSPACE 0x100
#define CRTPORT 0x3d4
static ushort *crt = (ushort*)0xb8000;  // CGA memory

static void
cgaputc(int c)
{
  int pos;

  // Cursor position: col + 80*row.
  outb(CRTPORT, 14);
  pos = inb(CRTPORT+1) << 8;
  outb(CRTPORT, 15);
  pos |= inb(CRTPORT+1);

  if(c == ’\n’)
    pos += 80 - pos%80;
  else if(c == BACKSPACE){
    if(pos > 0)
      crt[--pos] = ’ ’ | 0x0700;
  } else
    crt[pos++] = (c&0xff) | 0x0700;  // black on white

  if((pos/80) >= 24){  // Scroll up.
    memmove(crt, crt+80, sizeof(crt[0])*23*80);
    pos -= 80;
    memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
  }

  outb(CRTPORT, 14);
  outb(CRTPORT+1, pos>>8);
  outb(CRTPORT, 15);
  outb(CRTPORT+1, pos);
  crt[pos] = ’ ’ | 0x0700;
}

void
consputc(int c)
{
  if(panicked){
    cli();
    for(;;)
      ;
  }

  uartputc(c);
  cgaputc(c);
}
```

```
#define INPUT_BUF 128
struct {
  struct spinlock lock;
  char buf[INPUT_BUF];
  uint r;  // Read index
  uint w;  // Write index
  uint e;  // Edit index
} input;

#define C(x)  ((x)-'@')  // Control-x

void
consoleintr(int (*getc)(void))
{
  int c;

  acquire(&input.lock);
  while((c = getc()) >= 0){
    switch(c){
    case C('P'):  // Process listing.
      procdump();
      break;
    case C('U'):  // Kill line.
      while(input.e != input.w &&
            input.buf[(input.e-1) % INPUT_BUF] != '\n'){
        input.e--;
        consputc(BACKSPACE);
      }
      break;
    case C('H'):  // Backspace
      if(input.e != input.w){
        input.e--;
        consputc(BACKSPACE);
      }
      break;
    default:
      if(c != 0 && input.e-input.r < INPUT_BUF){
        input.buf[input.e++ % INPUT_BUF] = c;
        consputc(c);
        if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
          input.w = input.e;
          wakeup(&input.r);
        }
      }
      break;
    }
  }
  release(&input.lock);
}
```

```
int
consoleread(struct inode *ip, char *dst, int n)
{
  uint target;
  int c;

  iunlock(ip);
  target = n;
  acquire(&input.lock);
  while(n > 0){
    while(input.r == input.w){
      if(proc->killed){
        release(&input.lock);
        ilock(ip);
        return -1;
      }
      sleep(&input.r, &input.lock);
    }
    c = input.buf[input.r++ % INPUT_BUF];
    if(c == C('D')){  // EOF
      if(n < target){
        // Save ^D for next time, to make sure
        // caller gets a 0-byte result.
        input.r--;
      }
      break;
    }
    *dst++ = c;
    --n;
    if(c == '\n')
      break;
  }
  release(&input.lock);
  ilock(ip);

  return target - n;
}
```

```
int
consolewrite(struct inode *ip, char *buf, int n)
{
  int i;

  iunlock(ip);
  acquire(&cons.lock);
  for(i = 0; i < n; i++)
    consputc(buf[i] & 0xff);
  release(&cons.lock);
  ilock(ip);

  return n;
}

void
consoleinit(void)
{
  initlock(&cons.lock, "console");
  initlock(&input.lock, "input");

  devsw[CONSOLE].write = consolewrite;
  devsw[CONSOLE].read = consoleread;
  cons.locking = 1;

  picenable(IRQ_KBD);
  ioapicenable(IRQ_KBD, 0);
}
```

```
// Intel 8253/8254/82C54 Programmable Interval Timer (PIT).
// Only used on uniprocessors;
// SMP machines use the local APIC timer.

#include "types.h"
#include "defs.h"
#include "traps.h"
#include "x86.h"

#define IO_TIMER1       0x040           // 8253 Timer #1

// Frequency of all three count-down timers;
// (TIMER_FREQ/freq) is the appropriate count
// to generate a frequency of freq Hz.

#define TIMER_FREQ      1193182
#define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))

#define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
#define TIMER_SEL0      0x00    // select counter 0
#define TIMER_RATEGEN   0x04    // mode 2, rate generator
#define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first

void
timerinit(void)
{
  // Interrupt 100 times/sec.
  outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
  outb(IO_TIMER1, TIMER_DIV(100) % 256);
  outb(IO_TIMER1, TIMER_DIV(100) / 256);
  picenable(IRQ_TIMER);
}
```

```
# Initial process execs /init.

#include "syscall.h"
#include "traps.h"

# exec(init, argv)
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0  // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL

# for(;;) exit();
exit:
  movl $SYS_exit, %eax
  int $T_SYSCALL
  jmp exit

# char init[] = "/init\0";
init:
  .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
  .long init
  .long 0
```

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
```

```
// init: The initial user-level program

#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

char *argv[] = { "sh", 0 };

int
main(void)
{
  int pid, wpid;

  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0);  // stdout
  dup(0);  // stderr

  for(;;){
    printf(1, "init: starting sh\n");
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){
      exec("sh", argv);
      printf(1, "init: exec sh failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}
```

```
// Shell.

#include "types.h"
#include "user.h"
#include "fcntl.h"

// Parsed command representation
#define EXEC  1
#define REDIR 2
#define PIPE  3
#define LIST  4
#define BACK  5

#define MAXARGS 10

struct cmd {
  int type;
};

struct execcmd {
  int type;
  char *argv[MAXARGS];
  char *eargv[MAXARGS];
};

struct redircmd {
  int type;
  struct cmd *cmd;
  char *file;
  char *efile;
  int mode;
  int fd;
};

struct pipecmd {
  int type;
  struct cmd *left;
  struct cmd *right;
};

struct listcmd {
  int type;
  struct cmd *left;
  struct cmd *right;
};

struct backcmd {
  int type;
  struct cmd *cmd;
};
```

```
int fork1(void);  // Fork but panics on failure.
void panic(char*);
struct cmd *parsecmd(char*);

// Execute cmd.  Never returns.
void
runcmd(struct cmd *cmd)
{
  int p[2];
  struct backcmd *bcmd;
  struct execcmd *ecmd;
  struct listcmd *lcmd;
  struct pipecmd *pcmd;
  struct redircmd *rcmd;

  if(cmd == 0)
    exit();

  switch(cmd->type){
  default:
    panic("runcmd");

  case EXEC:
    ecmd = (struct execcmd*)cmd;
    if(ecmd->argv[0] == 0)
      exit();
    exec(ecmd->argv[0], ecmd->argv);
    printf(2, "exec %s failed\n", ecmd->argv[0]);
    break;

  case REDIR:
    rcmd = (struct redircmd*)cmd;
    close(rcmd->fd);
    if(open(rcmd->file, rcmd->mode) < 0){
      printf(2, "open %s failed\n", rcmd->file);
      exit();
    }
    runcmd(rcmd->cmd);
    break;

  case LIST:
    lcmd = (struct listcmd*)cmd;
    if(fork1() == 0)
      runcmd(lcmd->left);
    wait();
    runcmd(lcmd->right);
    break;
```

```
  case PIPE:
    pcmd = (struct pipecmd*)cmd;
    if(pipe(p) < 0)
      panic("pipe");
    if(fork1() == 0){
      close(1);
      dup(p[1]);
      close(p[0]);
      close(p[1]);
      runcmd(pcmd->left);
    }
    if(fork1() == 0){
      close(0);
      dup(p[0]);
      close(p[0]);
      close(p[1]);
      runcmd(pcmd->right);
    }
    close(p[0]);
    close(p[1]);
    wait();
    wait();
    break;

  case BACK:
    bcmd = (struct backcmd*)cmd;
    if(fork1() == 0)
      runcmd(bcmd->cmd);
    break;
  }
  exit();
}

int
getcmd(char *buf, int nbuf)
{
  printf(2, "$ ");
  memset(buf, 0, nbuf);
  gets(buf, nbuf);
  if(buf[0] == 0) // EOF
    return -1;
  return 0;
}
```

```
int
main(void)
{
  static char buf[100];
  int fd;

  // Assumes three file descriptors open.
  while((fd = open("console", O_RDWR)) >= 0){
    if(fd >= 3){
      close(fd);
      break;
    }
  }

  // Read and run input commands.
  while(getcmd(buf, sizeof(buf)) >= 0){
    if(buf[0] == ’c’ && buf[1] == ’d’ && buf[2] == ’ ’){
      // Clumsy but will have to do for now.
      // Chdir has no effect on the parent if run in the child.
      buf[strlen(buf)-1] = 0;  // chop \n
      if(chdir(buf+3) < 0)
        printf(2, "cannot cd %s\n", buf+3);
      continue;
    }
    if(fork1() == 0)
      runcmd(parsecmd(buf));
    wait();
  }
  exit();
}

void
panic(char *s)
{
  printf(2, "%s\n", s);
  exit();
}

int
fork1(void)
{
  int pid;

  pid = fork();
  if(pid == -1)
    panic("fork");
  return pid;
}
```

```
// Constructors

struct cmd*
execcmd(void)
{
  struct execcmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = EXEC;
  return (struct cmd*)cmd;
}

struct cmd*
redircmd(struct cmd *subcmd, char *file, char *efile, int mode, int fd)
{
  struct redircmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = REDIR;
  cmd->cmd = subcmd;
  cmd->file = file;
  cmd->efile = efile;
  cmd->mode = mode;
  cmd->fd = fd;
  return (struct cmd*)cmd;
}

struct cmd*
pipecmd(struct cmd *left, struct cmd *right)
{
  struct pipecmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = PIPE;
  cmd->left = left;
  cmd->right = right;
  return (struct cmd*)cmd;
}
```

```
struct cmd*
listcmd(struct cmd *left, struct cmd *right)
{
  struct listcmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = LIST;
  cmd->left = left;
  cmd->right = right;
  return (struct cmd*)cmd;
}

struct cmd*
backcmd(struct cmd *subcmd)
{
  struct backcmd *cmd;

  cmd = malloc(sizeof(*cmd));
  memset(cmd, 0, sizeof(*cmd));
  cmd->type = BACK;
  cmd->cmd = subcmd;
  return (struct cmd*)cmd;
}
```

```
// Parsing

char whitespace[] = " \t\r\n\v";
char symbols[] = "<|>&;()";

int
gettoken(char **ps, char *es, char **q, char **eq)
{
  char *s;
  int ret;

  s = *ps;
  while(s < es && strchr(whitespace, *s))
    s++;
  if(q)
    *q = s;
  ret = *s;
  switch(*s){
  case 0:
    break;
  case ’|’:
  case ’(’:
  case ’)’:
  case ’;’:
  case ’&’:
  case ’<’:
    s++;
    break;
  case ’>’:
    s++;
    if(*s == ’>’){
      ret = ’+’;
      s++;
    }
    break;
  default:
    ret = ’a’;
    while(s < es && !strchr(whitespace, *s) && !strchr(symbols, *s))
      s++;
    break;
  }
  if(eq)
    *eq = s;

  while(s < es && strchr(whitespace, *s))
    s++;
  *ps = s;
  return ret;
}
```

```
int
peek(char **ps, char *es, char *toks)
{
  char *s;

  s = *ps;
  while(s < es && strchr(whitespace, *s))
    s++;
  *ps = s;
  return *s && strchr(toks, *s);
}

struct cmd *parseline(char**, char*);
struct cmd *parsepipe(char**, char*);
struct cmd *parseexec(char**, char*);
struct cmd *nulterminate(struct cmd*);

struct cmd*
parsecmd(char *s)
{
  char *es;
  struct cmd *cmd;

  es = s + strlen(s);
  cmd = parseline(&s, es);
  peek(&s, es, "");
  if(s != es){
    printf(2, "leftovers: %s\n", s);
    panic("syntax");
  }
  nulterminate(cmd);
  return cmd;
}

struct cmd*
parseline(char **ps, char *es)
{
  struct cmd *cmd;

  cmd = parsepipe(ps, es);
  while(peek(ps, es, "&")){
    gettoken(ps, es, 0, 0);
    cmd = backcmd(cmd);
  }
  if(peek(ps, es, ";")){
    gettoken(ps, es, 0, 0);
    cmd = listcmd(cmd, parseline(ps, es));
  }
  return cmd;
}
```

```
struct cmd*
parsepipe(char **ps, char *es)
{
  struct cmd *cmd;

  cmd = parseexec(ps, es);
  if(peek(ps, es, "|")){
    gettoken(ps, es, 0, 0);
    cmd = pipecmd(cmd, parsepipe(ps, es));
  }
  return cmd;
}

struct cmd*
parseredirs(struct cmd *cmd, char **ps, char *es)
{
  int tok;
  char *q, *eq;

  while(peek(ps, es, "<>")){
    tok = gettoken(ps, es, 0, 0);
    if(gettoken(ps, es, &q, &eq) != 'a')
      panic("missing file for redirection");
    switch(tok){
    case '<':
      cmd = redircmd(cmd, q, eq, O_RDONLY, 0);
      break;
    case '>':
      cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
      break;
    case '+':  // >>
      cmd = redircmd(cmd, q, eq, O_WRONLY|O_CREATE, 1);
      break;
    }
  }
  return cmd;
}
```

```
struct cmd*
parseblock(char **ps, char *es)
{
  struct cmd *cmd;

  if(!peek(ps, es, "("))
    panic("parseblock");
  gettoken(ps, es, 0, 0);
  cmd = parseline(ps, es);
  if(!peek(ps, es, ")"))
    panic("syntax - missing )");
  gettoken(ps, es, 0, 0);
  cmd = parseredirs(cmd, ps, es);
  return cmd;
}

struct cmd*
parseexec(char **ps, char *es)
{
  char *q, *eq;
  int tok, argc;
  struct execcmd *cmd;
  struct cmd *ret;

  if(peek(ps, es, "("))
    return parseblock(ps, es);

  ret = execcmd();
  cmd = (struct execcmd*)ret;

  argc = 0;
  ret = parseredirs(ret, ps, es);
  while(!peek(ps, es, "|)&;")){
    if((tok=gettoken(ps, es, &q, &eq)) == 0)
      break;
    if(tok != ’a’)
      panic("syntax");
    cmd->argv[argc] = q;
    cmd->eargv[argc] = eq;
    argc++;
    if(argc >= MAXARGS)
      panic("too many args");
    ret = parseredirs(ret, ps, es);
  }
  cmd->argv[argc] = 0;
  cmd->eargv[argc] = 0;
  return ret;
}
```

```
// NUL-terminate all the counted strings.
struct cmd*
nulterminate(struct cmd *cmd)
{
  int i;
  struct backcmd *bcmd;
  struct execcmd *ecmd;
  struct listcmd *lcmd;
  struct pipecmd *pcmd;
  struct redircmd *rcmd;

  if(cmd == 0)
    return 0;

  switch(cmd->type){
  case EXEC:
    ecmd = (struct execcmd*)cmd;
    for(i=0; ecmd->argv[i]; i++)
      *ecmd->eargv[i] = 0;
    break;

  case REDIR:
    rcmd = (struct redircmd*)cmd;
    nulterminate(rcmd->cmd);
    *rcmd->efile = 0;
    break;

  case PIPE:
    pcmd = (struct pipecmd*)cmd;
    nulterminate(pcmd->left);
    nulterminate(pcmd->right);
    break;

  case LIST:
    lcmd = (struct listcmd*)cmd;
    nulterminate(lcmd->left);
    nulterminate(lcmd->right);
    break;

  case BACK:
    bcmd = (struct backcmd*)cmd;
    nulterminate(bcmd->cmd);
    break;
  }
  return cmd;
}
```